Design and Implementation of a Real-Time Video and Graphics Scaler

by

Erika Shu-Ching Chuang

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 20, 1997

Author .....
Department of Electrical Engineering and Computer Science
May 20, 1997

Certified by .....
V. Michael Bove
Associate Professor of Media Technology, MIT Media Laboratory
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Design and Implementation of a Real-Time Video and Graphics Scaler

by

Erika Shu-Ching Chuang

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Various methods for scaling video and graphics images were studied in depth. The evaluation and selection of the optimal algorithms involved comparing the quality of scaled images, analyzing the hardware constraints imposed by the video and graphics architecture, and estimating the complexity and cost of the implementations. A real-time hardware scaler was designed using the chosen algorithms: Linear Interpolation and Area Weighted Averaging. The behavioral and register transfer level description of the hardware design was created and simulated with test benches.

Thesis Supervisor: V. Michael Bove
Title: Associate Professor of Media Technology, MIT Media Laboratory

Thesis Supervisor: Mark Kautzman
Title: Staff Engineer, IBM Microelectronics

Thesis Supervisor: Mark Merrill
Title: Manager, IBM Microelectronics

Acknowledgments

I would like to thank all the members in the Display Subsystems Solutions group at IBM Microelectronics from whom I have received assistance. I would like to thank Mark Kautzman for introducing me the good graphics and image processing books and many other useful resources. I am grateful to the team of engineers, Tom Wilson, John Pike, Hiro Ando, Pete Casavant, Paul Schanely, and Bob Horton, who gave me a lot of good ideas in hardware design and answered my endless questions. I would like to thank Mike West for his awesome computer graphics notes. I would like to thank my manager Mark Merrill for helping me find my thesis topic and putting up with my strange working schedule.

I am grateful to Professor V. Michael Bove for giving me useful pointers in my research.

I would like to thank Ya-Chieh Lai, Theodore Tonchev, and Andrew Chen for helping me with my thesis writing.

Foremost, I would like to thank my family for their love and support. Without them I would not have reach where I am today.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With advances in silicon technology and the development of the high-bandwidth peripheral component interconnect (PCI) bus, the fusion of multimedia with the traditional personal computer has become possible. People can now buy personal computers that combine real-time audio processing, display and manipulation of video data, or play computer games with high quality three dimensional graphics rendering. When using graphical user interfaces (GUIs), users may want to display several windows at a time or scale the windows to arbitrary sizes. Therefore the ability to scale images in real-time is very important in the modern video and graphics systems.

Real-time video scaling can be achieved by either software or hardware. The software approach typically results in a combination of low image quality and slow update times. High quality video scaling is possible in software but cannot be performed in real time. The hardware approach has the advantage that it can process large amounts of data when the control is relatively simple. However, implementing a high quality video scaler in a VLSI architecture poses several challenges. These challenges are a result of a limited hardware resources, such as limited memory bandwidth and silicon area. The difficulty in real time image scaling is that the source information needs to be preserved as much as possible. At the same time, the signal needs to be properly bandlimited to comply with the Nyquist sampling rate criterion, which requires some digital filtering.

The goal of this project is to investigate various scaling algorithms and the issues involved in implementing them in hardware. There are three main criteria for the design of the scaler hardware: quality, performance, and cost. Ideally, we would like the scaled video or graphic images be clear and relatively free of aliasing. At the same time, the scaler must be compatible with existing video and graphics system architecture and be economically feasible. The existing video and graphic system puts some constraints and requirements on the scaler hardware. First, the scaler must load pixel data from the graphic memory, perform the scaling in real time, and then output the data to the screen. The scaler can only operate with limited data bandwidth; at the same time, the scaler must

meet the display frequency requirement for a wide range of CRT controllers. The goal for this project is to achieve the best possible image quality given the hardware constraints. After comparing the trade-off between scaled image quality and the complexity in hardware implementations, optimal algorithms are selected and implemented in Verilog Hardware description language and simulated using software tools.

1.1 Outline of the Thesis

Chapter 2 begins with the concept of image scaling, which is described in terms of the basic sampling theory in digital signal processing. The second section introduces various methods for image scaling. These methods are implemented in C and performed on a set of sample still images and moving image sequences. The resulting scaled still images and image sequences are displayed side by side for comparison. Chapter 3 discusses the hardware requirements and constraints in a typical video/graphics system architecture. The algorithms introduced in Chapter 2 are evaluated in terms of complexity and feasibility for hardware implementation. In Chapter 4, optimal methods are selected and the detailed algorithms are described in pseudo codes. Chapter 5 describes the architecture of the scaler, the functionality and implementation of its basic modules. The last chapter includes a summary and describes work that needs to be done in the future.

Chapter 2

Background

2.1 DSP Basic and Sampling Rate Conversion

One of the key concepts in signal processing is the sampling of a continuous time signal. Figure 2.1 shows the frequency spectrum of a bandlimited analog signal x(t), which has no energy above frequency B. Figure 2.2 shows the frequency response of sampled x(t). The baseband is repeated at integer multiples of the sampling frequency ω_s. The Nyquist theorem states that if a signal x(t) is bandlimited to B, then x(t) can be reconstruct from its samples x[n], if $\omega_s > 2B$.

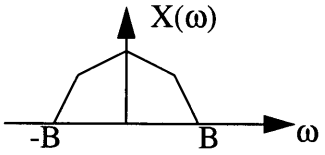


Figure 2.1: Frequency spectrum of bandlimited analog signal x(t)

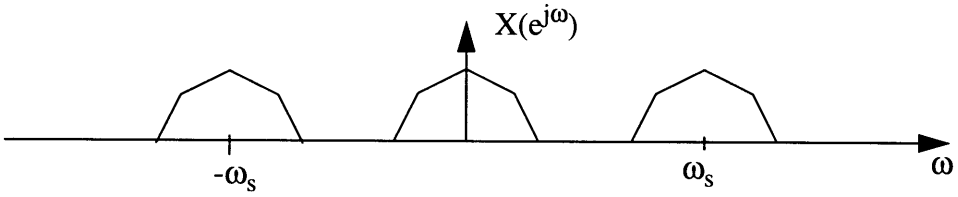


Figure 2.2: Spectrum of sampled x(t) at ω_s, where $\omega_s = 2\pi/T$

Scaling video or graphic images in real time is a multirate DSP problem ([1], [2]). In this case the input signal as already been sampled at a sampling period T (determined by the input image resolution) and the goal is to convert it to a different sampling period T' (determined by the output image resolution). The process of decreasing the sampling rate is called decimation. The process of increasing the sampling rate is called interpolation. In graphics terms, the period is δx because it is in the spatial domain as opposed to time domain. When $\delta x' < \delta x$, the image is being enlarged, and when $\delta x' > \delta x$, the image is being reduced. Both interpolation and decimation introduce unwanted high frequency components that need to be removed with low-pass filtering. Figure 2.3 - 2.6 illustrate this

concept. Figure shows the process of increasing the sampling rate by a factor of 2. Figure a is a section of the original data. The data first goes through a sampling rate expansion state (figure b), where zeros are padded in between the original samples. The interpolation procedure, shown in figure c, estimates the values between the original samples. Figure 2.3 illustrates the same steps in frequency domain. The process of sampling rate expansion scales the frequency axis and results in extra copies of the original spectrum in the high frequency range (figure 2.3 b). The interpolation filter removes the high frequency components, leaving one scaled version of the original spectrum (figure 2.3 c).

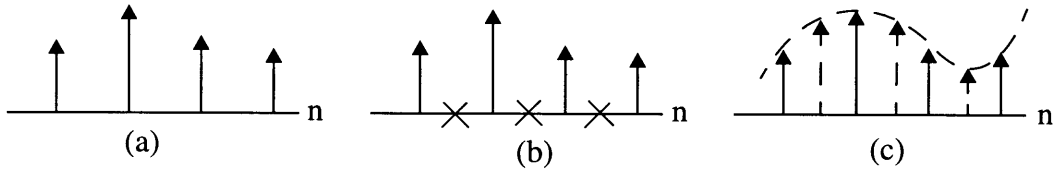


Figure 2.3: Interpolation by a factor of 2. (a) a section of original samples (b) sampling rate expansion (c) interpolation

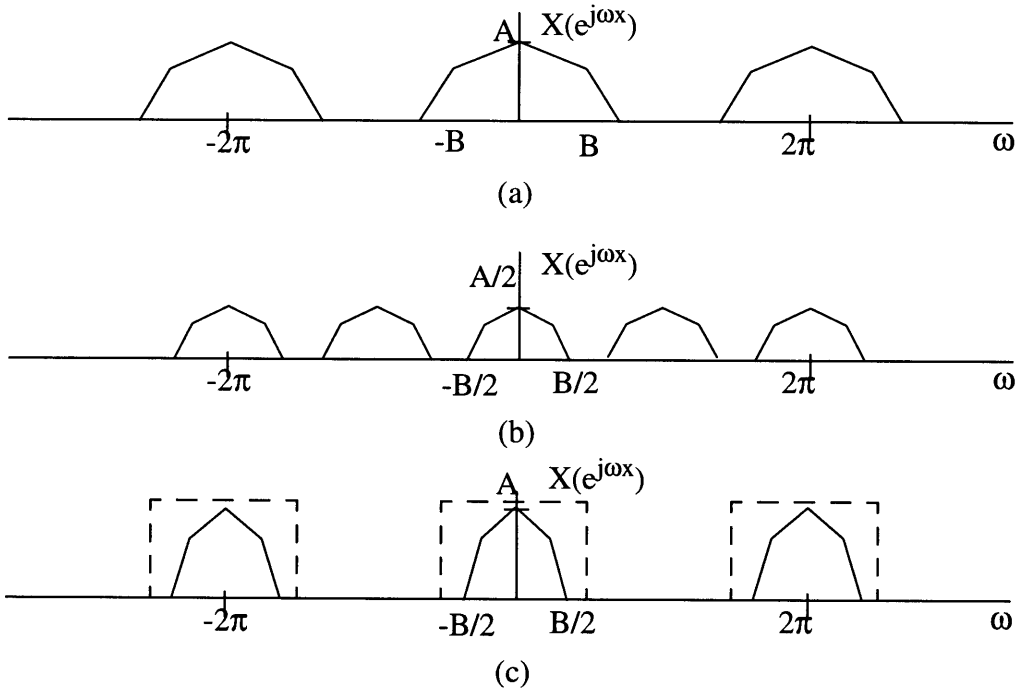


Figure 2.4: Interpolation by a factor of 2- frequency domain representation. The figures correspond to the same steps in figure 2.3. (a) original spectrum (b) sampling rate expansion(c) interpolation

Figure 2.5 shows the process of decreasing the sampling rate by a factor of 2. The process of decimation scales the frequency axis. The resulting spectrum of the original data is stretched and may result in aliasing if the original data is not bandlimited (as shown in figure 2.6b). Therefore, a good image reduction algorithm involves in removing the high frequency components before decimation.

Figure 2.5: Downsampling by a factor of 2. (a) a section of original samples (b) decimation

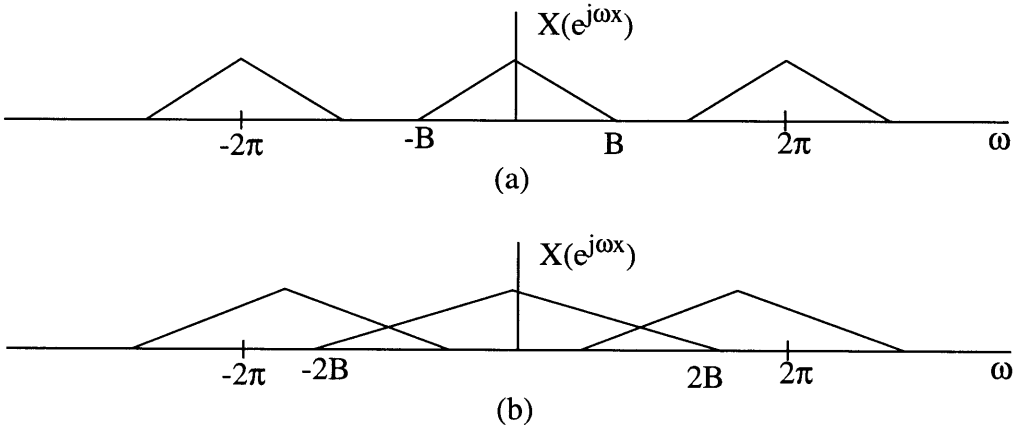


Figure 2.6: Downsampling by a factor of 2 - frequency domain representation (a) original spectrum (b) decimation with high frequency aliasing.

In conclusion, both image enlargement and reduction involve low-pass filtering. The quality of most scaling methods, regardless of their complexities, will be a function of the frequency response of the filter.

2.2 Methods for Image Scaling

The next section will introduce commonly used scaling methods. There are many other methods for image scaling, such as polynomial interpolation [6] and Lagrange Interpolation [1]. However, because of high computational demands or difficulty of hardware

implementation, these methods are not investigated in the thesis.

2.2.1 Anti-Aliased Resampling

In an ideal scaler, the frequency content scales proportionally with the image size, both horizontally and vertically. Figure 2.7 illustrates the process of anti-aliased resampling ([1],[2]). The input data is upsampled by a factor of L and downsampled by a factor of M, where L and M are integers. The scaling operation can be divided into two stages. In the first stage (interpolation), the input signal is processed by a sampling rate expander [1], which inserts samples of zero value in between the input samples. High-frequency noise created by this interpolation process is removed by a lowpass filter with a cutoff frequency of π/L. In the second stage (decimation), the intermediate data is prefiltered with a cutoff frequency of π/M to remove the image frequencies above Nyquist rate. It is then processed by a sampling rate compressor, which discards samples to achieve the desired sampling rate. The ratio of M/L determines the scaling factor. Figure 2.8 shows the simplified system in which the decimation and interpolation filters are combined. The resulting lowpass filter has a gain of L and a cutoff frequency which is the minimum of π/L and π/M.

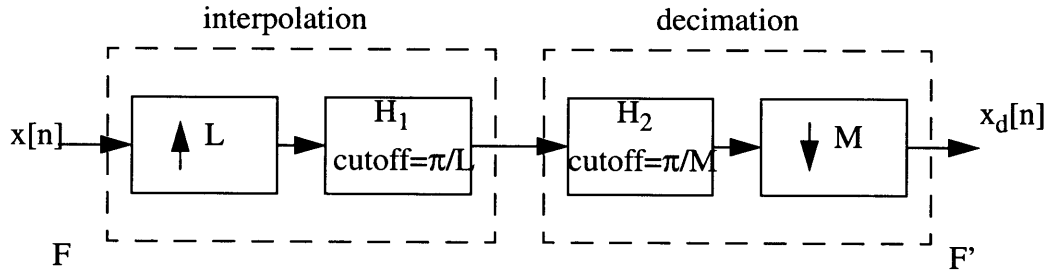


Figure 2.7: Block diagram for anti-aliased resampling ([1],[2])

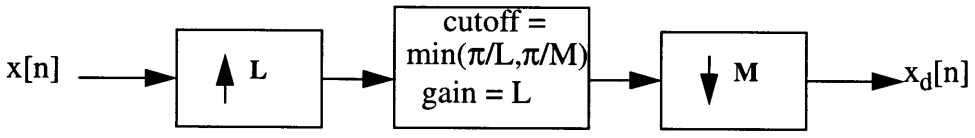


Figure 2.8: Block diagram for simplified anti-aliased resampling ([1],[2])

The resampling method discussed above works for all rational number scaling factors by mapping F pixels to F' pixels. However, when F and F' are relatively prime, L and M may be very large. There may not be enough memory to hold the large number of upsampled data at the sample rate expander stage, or enough computation cycles to properly filter the intermediate output. An alternative of this type of design is achieved with a "polyphase" filter [1]. This method combines the filter and the resampler by switching its coefficients as the relative position of input and output samples changes. This method is difficult to implement when the scaling factor is arbitrary and is rarely used in hardware implementation.

2.2.2 Pixel Dropping and Duplication

The simplest form of scaling down is pixel dropping, where M out of every N pixels are thrown away both horizontally and vertically. For example, a scaling factor of two thirds results in one out of every three pixels being discarded in both the horizontal and vertical directions. Simple upsampling can be accomplished by pixel duplication, where M out of every N pixels are duplicated both horizontally and vertically. The most common algorithms used to determine which pixels to duplicate and which pixels to drop are the line drawing algorithms, such as Digital Differential Analyzer algorithms or the Bresenham algorithm ([3],[4],[5]). Thesis methods are easy and inexpensive to implement in real time. However, dropping pixels results in the loss of detail and introduces aliasing components, while duplicating pixels result in blocks within the image and is unacceptable when the scale factor is large.

2.2.3 Linear Interpolation

An improvement in scaled image quality can be achieved using bilinear interpolation ([6], [7]). It is also called bilinear interpolation if the process combines both the horizontal vertical dimensions. When an output sample falls between two input samples, the output sample is computed by the equation:

$$P_{out}(m) = w(m,n)P_{in}(n+1) + (1 - w(m,n))P_{in}(n) \tag{2.1}$$

where

$$w(m, n) = \frac{N-1}{M-1} \times m - n,$$

and N and M are the original and desired dimension of the image respectively. However, reducing images size to smaller than one half still results in aliasing due to deleted pixels, and excess high-frequency detail is removed unnecessarily in scaling up.

2.2.4 Pre- and Postfiltering

Another way of filtering is to apply fixed filter coefficients with a resampler [7]. The order in which filters are applied depends on whether the image is being enlarged or reduced. When reducing an image, the decimation filter is applied first, which bandwidth-limits the image horizontally and vertically before the pixel dropping by spreading out the contribution of each source pixel to adjacent source pixels. When enlarging an image, the interpolation filter is applied after scaling to smooth out the rough edges resulting from duplicated blocks of pixels. In general, the higher the filter quality, the larger the number of taps in the filter. Furthermore, each scaling factor requires a different set of filter coefficients. The simplest approach for the pre-and postfiltering is to use a number of pre-calculated filter coefficients. The disadvantage is that it only works well for certain scaling factors. A common implementation is a three-tap smoothing filter that averages the current input pixel with two adjacent pixels (see table 2.1 [5]). This method is easy to implement, however, for scaling up more than about two times, the simple three-tap filter can no longer reduce the blocks resulting from duplicating the pixels. When reducing the image size to more than about one half, the aliasing introduced can no longer be minimized. Furthermore, the resulting image would sometimes look too blurry and additional sharpening filter might be required either before or after the scaling. Better filtering functions could be obtained by using a five-tap filter or more sophisticated FIR filter, but the logic required for the filtering will increase accordingly.

Table 2.1: Simple three-tap smoothing filters

Degree of smoothing	Moderate	Aggressive
Coefficients	(1/4, 1/2, 1/4)	(1/2, 0, 1/2)

2.2.5 Anti-aliased interpolation

Another approach uses a combined filter/interpolator followed by a resampler [3]. The center of the filter transfer function always aligned over the output grid. For each scaling factor, the filter transfer function is stretched to remain aligned over the output samples. The generalized equation for anti-aliased interpolation follows:

$$P_{out}(m) = \frac{1}{S}\sum_{i \in N} w(m, n, i)P_{in}(n + i) \tag{2.2}$$

Where *w(m,n,i)* is the input pixel weight, which is a function of the relative input and output pixel position, filter function, and scaling factor. *S* is the normalizing factor. For image enlargement, *S*=1, and for image reduction, S = scaling factor. The total number of contributing factors, *N*, also depends on the filter shape and scaling factor. For image enlargement N is constant, while for image reduction N becomes bigger as the shape of the interpolating filter is stretched.

This general concept can be applied with different filter functions to achieve better frequency responses. Furthermore, anti-aliasing can be performed by stretching the filter shape to include more input samples. Many filter functions have been proposed in literature. Ideally, the *sinc* function provides perfect interpolation, but it uses infinitely many samples and cannot be carried out in practice. One can spatially limit the *sinc* function by truncating its tails. However, a simple truncation leads to Gibbs phenomenon, which manifests itself as ripples in the frequency behavior of the filter in the vicinity of filter magnitude discontinuities. If the *sinc* function is truncated, the amplitude of the ripples do not decrease even as the duration of the filter increases. This problem can be minimized with a windowed *sinc* function, in which the slopes at the ends of the waveform are forced to zero

to decrease the discontinuity. Although the windowed *sinc* functions are useful, they are relatively expensive because the window must be fairly wide. A variety of other functions are often used instead. The most common examples are bilinear, biquadratic, and bicubic interpolators. [3]. In general, as the order of interpolation increases, the scaled image becomes smoother, but high frequency components are lost so that the picture tends to look blurry. Figure 2.9 shows some common functions for the interpolator. For image reduction, the widths of the functions are increased, and the heights are scaled proportionally.

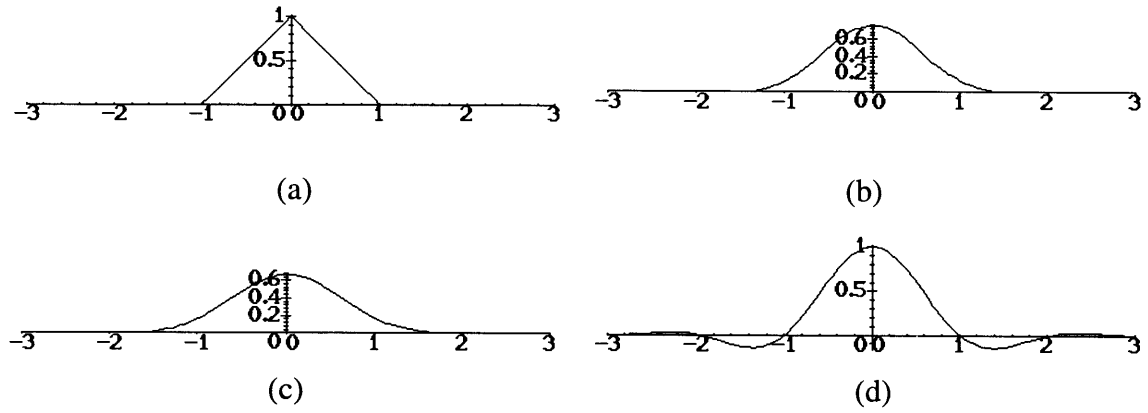
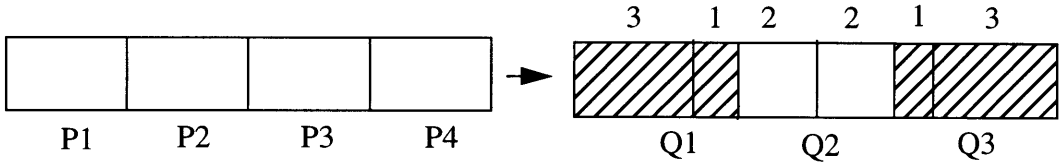


Figure 2.9: Common functions for interpolators [3]: (a) bilinear (b) biquadratic (c) bicubic (d) Lanczos interpolator

2.2.6 Area Averaging

A common approach of area averaging to is to give each pixel in the original picture an equal weight for the pixels in the scaled picture [4]. This method is often used in for image reduction. Figure 2.10 shows an example of reducing 4 input pixels to 3 output pixels; each input pixel contributes 3/4 of its value to an output pixel. One can think of this method as a hybrid of a zero order (box) and first order (bilinear) anti-aliased interpolator such that when the reduction factor is small, the filter assigns linear weights to contributing input pixels; when the reduction factor is larger, the filter assigns the same weight to all contributing input pixels, with the exception of the pixels at the two ends.



$$Q1 = \frac{(3P1 + 1P2)}{4}$$

$$Q2 = \frac{(2P2 + 2P3)}{4}$$

$$Q3 = \frac{(1P3 + 3P4)}{4}$$

Figure 2.10: Area Weighted Averaging (4:3 reduction)

2.3 Comparison of the Software Methods

The scaler must be able to handle both natural and created images. Natural images can be sourced from either a still photograph, or a sequence of image frames reconstructed from a compressed source (e.g. from MPEG). Created images may come from animated applications, such as video games with 3D graphics rendering. Natural image sequences are in interlaced format, at 30 frames per second (60 fields per second), and typically have input resolutions from 352 x 240 (MPEG1) to 720 x 480 (MPEG2) in the YUV (also called Y CbCr) color space. The still images are in 8:8:8 RGB format. A set of still images and MPEG image sequences are used in the comparison of these software implementations of the algorithms. The MPEG are standard text sequences in subsampled 4:2:0 and 4:2:2 format. The differences between these color-component formats, as well as the hardware implication for these differences, will be discussed in section 5.2.1 (Pixel Format Unit). For this part of the project, the subsampling in the YUV data is removed, and color space conversion is performed such that all scaling algorithms are applied in RGB color space.

A test environment is setup where side by side comparisons can be made on still images and image sequences. The program that implements various scaling algorithms are written in C; it performs scaling for horizontal and vertical directions in two passes (see Appendix A). A few examples are shown here. Figure 2.11 shows one frame of the standard MPEG1 image (352x240) 'Flower Garden', enlarged by a horizontal factor of 600/352, and a vertical factor of 410/240. The image scaled with the Bresenham algorithm, as

(a)

(b)

Figure 2.11: Image enlargement: (a) Bresenham Algorithm (b) Bilinear Interpolation

shown in figure 2.11a, has rough edges that can be observed at the poles of the street lamps, the roof of the house, and the branches of the tree. Sequences of images produced by Bresenham algorithm have artifacts that resemble looking at the picture through pattern distorted glass. Bilinear interpolation removes some of these artifacts and produces smoother image, as shown in figure 2.6 (b). For scaling factors larger than 2, bilinear interpolation becomes insufficient and more sophisticated filtering is required to remove the artifacts. In general, the design of a resampling filter involves trade-off between the sharpness of the picture (leakage) and the ringing effect (Gibbs phenomenon). For example, the higher order interpolators (e.g. bicubic) produce smooth but somewhat blurred scaled images. More sophisticated filter functions preserve the sharp edges by using larger widths and negative coefficient values. For example, the Lanczos interpolator is the *sinc* function with tapered ends, and can be approximated is a third order polynomial with negative values at the two ends. However, in a practical system, all of the filtering must be performed with reasonable amount of computation. Most of the high end scaling algorithms will be too costly for hardware implementation.

Good quality image reduction involves filtering high frequency components before downsampling to avoid aliasing. Figure 2.12 is an image of a man wearing a striped shirt, which suffers severely to aliasing in image reduction. Figure 2.13 shows the images after being reduced by 20% using Bresenham algorithm and Area-weighted Averaging. Note the severe aliasing in the shirt when pixels are dropped without any averaging, as show in figure 2.13a. The Area-weighted averaging reduces the aliasing, as shown in figure 2.13b. When the size of the picture is reduced further, the amount of details will be lost through the averaging. The loss is irreversible, but it will not be a major issue because the image is being displayed on the screen without further processing. Higher order filtering may be able to preserve more details, but for many users, image reduction is not as important because the picture will be too small to be seen clearly.

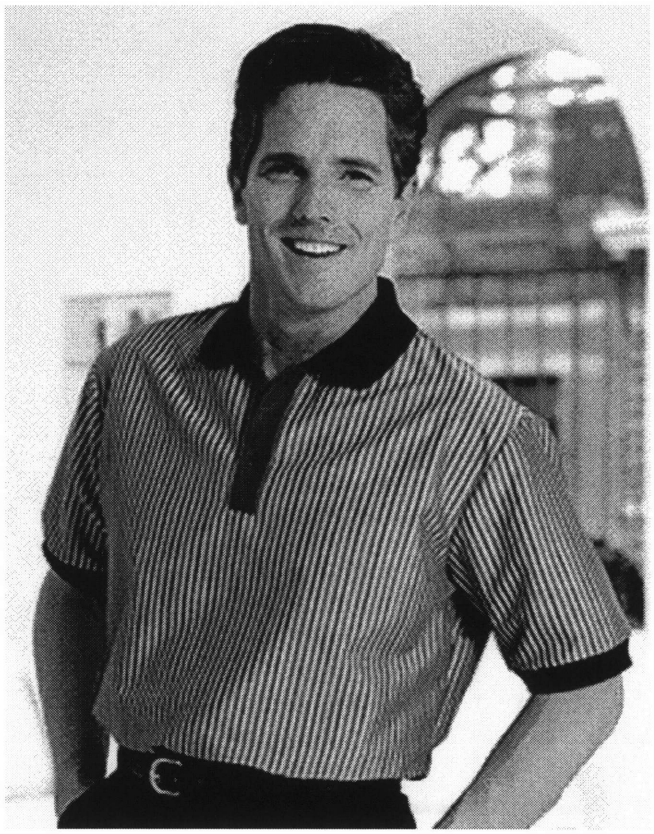
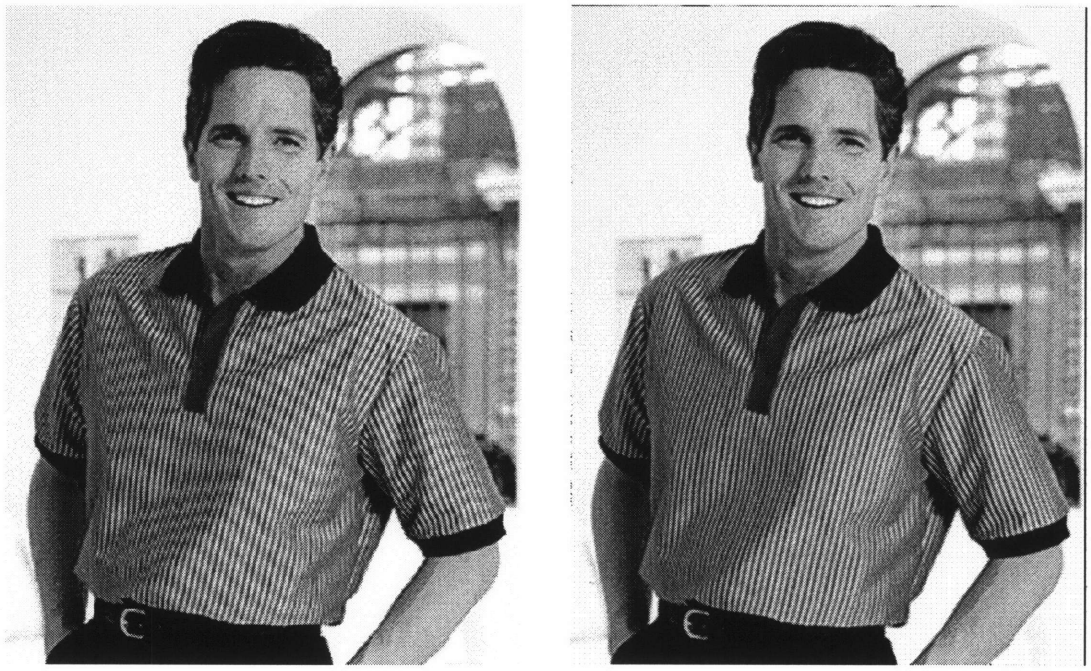


Figure 2.12: Original Image of man in striped shirt



(a) (b)

Figure 2.13: Image of man in striped reduced by 20% with (a) Bresenham algorithm (b) Area Weighted Averaging

Chapter 3

Design Constraints

This chapter discusses some of the many issues which influence the design of the video/ graphics scaler in order to produce a viable product: graphics memory bandwidth, pixel processing requirements, and hardware cost. This chapter begins by describing the video system hardware, which provides the background for understanding how the scaler integrates into the overall system. Graphics memory bandwidth and pixel processing requirements and are two of the obstacles need to be overcome to meet the demands of today's high-performance graphics. The last section will discuss the components that constitute the major cost of implementing a hardware scaler.

3.1 Video/Graphics System Hardware Overview

The functions of the video/graphics system can be divided into two categories: update graphics and display graphics [8]. Update graphics is concerned with the processes of entering or modifying data in the graphics system. Display graphics is concerned with the processes of getting data out of the graphics system and onto the screen of the display device. Figure 3.1 shows a commonly used one-port architecture for video and graphics systems. In this type of architecture, video and graphics data use different region of the same physical memory. One disadvantage in this design is that the update graphics functions and the display graphics functions share the same graphics memory port and contend for the bandwidth available at that port. A number of different types of memory device are used to form one-port graphics architectures. These memory devices include DRAM (Dynamic Random Access Memory), SDRAM (Synchronous DRAM), and SGRAM (Synchronous Graphics RAM).

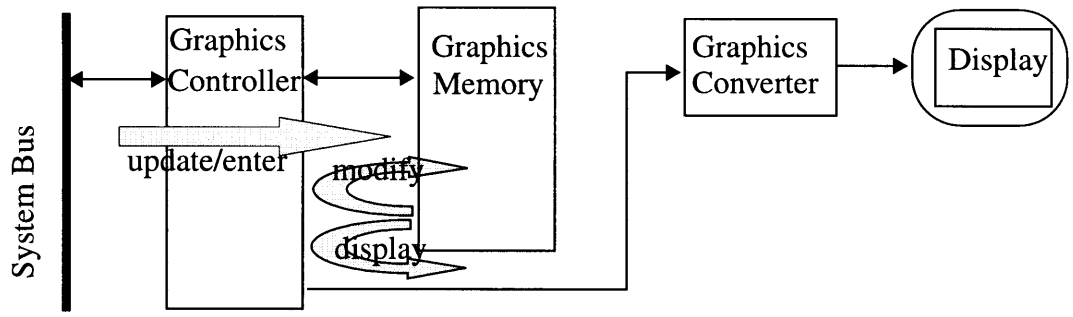


Figure 3.1: Example Architecture of a one-port Graphics System [8].

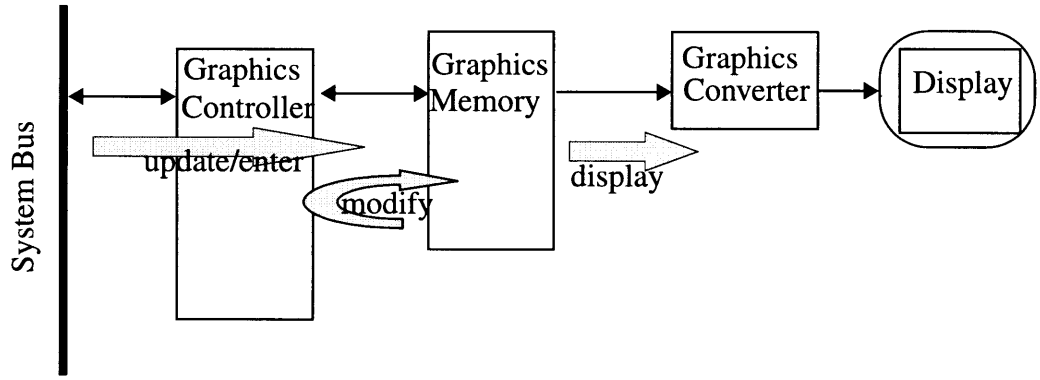


Figure 3.2: Example Architecture of a two-port Graphics System [8].

Two-port graphics architectures (shown in figure 3.2) are also used in many systems. One port is dedicated for outputting the display data, and the other port is used for update access. The most common memory device used in a two-port graphics architecture is a VRAM (Video Random Access Memory). In general, VRAMs offer higher performance at a higher cost. However with advances in semiconductor technology and strong competition in the graphics market, one-port architectures have been steadily increasing their performance and function while still maintaining their lower cost, giving two-port architecture much pressure [8]. The design for this project uses a one-port memory (SGRAM) architecture.

3.2 Real-time Scaler

The video and graphics scaler must perform real-time scaling for both decoded MPEG

video streams and rendered 3D images while the images are being outputted to the display pipeline (figure 3.3). Therefore the constraint at the input of the scaler is the bandwidth of the data bus from the video/graphics memory to the scaler. This bandwidth restricts the speed at which the scaler can access the video/graphics memory. The constraint at the output of the scaler is the display pixel frequency.

3.2.1 Graphics memory bandwidth

The memory used for this project is SGRAM (synchronous graphics random access memory). It has a memory bus width of 64 bits, and can operate at 100 MHz. When operated in burst mode, these devices can read or write an amount of data equal to their bus width every clock cycle. For a 64 bit data bus running at 100 MHz, the peak bandwidth will be 800 Mbytes/sec. However, the efficiency of the bus depends on the design of the graphics controller and its ability to operate in page mode as much as possible. In a one-port graphics architecture, all update graphics functions and display graphics functions share the same port and cause bus contention. Some of these functions include the display data, video decoding, 3D rendering, etc.

3.2.2 Pixel processing requirement

The output pixel frequency of the scaler must match the display frequency to guarantee a continuous, non-flickering image on the PC desktop monitor. The pixel frequency (f_p) can be approximated by following equation (assuming a progressive scanned display device):

$$f_p = \left(\frac{P}{HADT}\right)x\left(\frac{L}{VADT}\right)xF \tag{3.1}$$

where

- P = pixels per line
- L = number of lines
- F = refresh rate
- $HADT$ = Percentage Horizontal Active Display Time (for CRTs, it is the percentage of the frame time the beams are not blanked)
- $VADT$ = Percentage Vertical Active Display Time.

For most CRTs today, the output display resolution varies from 640 x 480 to 1600 x 1200,

the refresh rate varies from 60 to 90 frames per second, and the total active display time (*ADT = HADTxVADT)* ranges from 60% to80%. This results in an output pixel rate from 25 Mpixels per second to approximately 250 Mpixels/s. The table below shows pixel frequencies for a number of popular CRT display modes at different refresh rates and with some assumptions made about *ADT*. These numbers provide an idea of the ranges of pixel frequency necessary for a given display resolution.

Table 3.1: Pixel frequency as a function of display addressability, refresh rate, and active display time [8].

Resolution (*PxL*)	**# Pixels**	**f_p 60Hz / 75%**	**f_p 70Hz / 70%**	**f_p 85Hz / 65%**
640 x 480	307,200	25 MHz	31MHz	40 MHz
800 x 600	480,000	38 MHz	48 MHz	63 MHz
1024 x 768	768,432	61 MHz	77 MHz	100 MHz
1280 x 1024	1,310,720	105 MHz	131 MHz	171 MHz
1600 x 1200	1,920,000	154 MHz	192 MHz	251 MHz

In most algorithms, each pixel in the output image is a weighted sum of the neighboring pixels in the source image (determined by the horizontal and vertical filter widths). Because the original image is in raster format, pixels from the same vertical line are not available simultaneously. Display images are normally stored in a raster format and accessed on a line by line basis. Thus, scaling in the vertical direction poses a very different challenge from scaling in the horizontal direction. Horizontal scaling is relatively straight forward because the delay elements are simply registers. Vertical filtering, on the other hand, typically requires an expensive line store for each filter tap. The vertical filter width determines the number of lines that need to be accessed by the scaler from the graphic memory for a given output line. Figure 3.3 shows example of resizing an resource image of size n to size n'. δT_h is the total horizontal time ($\delta T = \frac{p}{HADTxf_s}$), and *WinAct* is the horizontal window active time for the output image($WinAct = \frac{n'}{f_s}$). While a true real-time scaling requires the memory access, scaling and outputting a line to be performed in *WinAct*, in reality the constraint is usually the less strict δT.

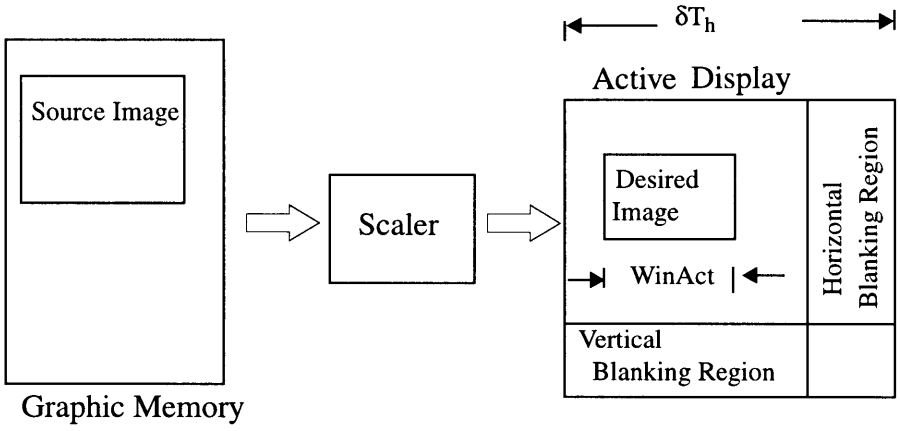


Figure 3.3: Illustration of the data flow requirement.

The memory bandwidth requirement for a particular vertical filter of N taps can be approximated by the following equation.

$$B = \frac{n \times N \times S}{\delta T_h} \tag{3.2}$$

where

- n= number of pixels in a line in original image
- N = width of vertical filter (number of vertical filter taps)
- S = pixel size (bytes)

The pixel processing requirement and the limited amount of memory bandwidth puts a limitation on the vertical filter width. Since the display subsystem needs to provide a continuous flow of pixel data to the monitor, the display graphics functions generally (e.g. the scaler) have higher priorities for utilizing this limited bandwidth than the update graphics functions. However, if the scaler alone requires too much bandwidth, the performance of all the other functions in update graphics will degrade accordingly. This constraint is important in choosing a scaling algorithm in the vertical direction.

3.2.3 Hardware cost

In today's competitive graphics market, quality and cost determine the consumer demand of any product. Therefore trade-off must be made between these two factors in deciding which implementation best suits the design purpose. The cost of a design is determined by

many factors including complexity of the control logic, size and precision of the arithmetic unit, external/internal datapath width, amount of storage internal memory store.

Arithmetic unit

The input data stream is filtered in some fashion regardless of the algorithm involved. Figure 3.4 is a hardware implementation of a classical FIR filter, called a tapped delay line. It requires one accumulator and N multipliers for an N-tap filter. Since multipliers are much more costly than accumulators, this implementation is generally not desirable for wide filters. If processing speed allows, one multiply and accumulate unit can be used for multiple taps. Furthermore, in order to achieve the same margin of error, higher order filters generally require greater precision of the filter coefficients and the intermediate values, requiring greater precision in representing coefficients and wider multipliers and accumulators.

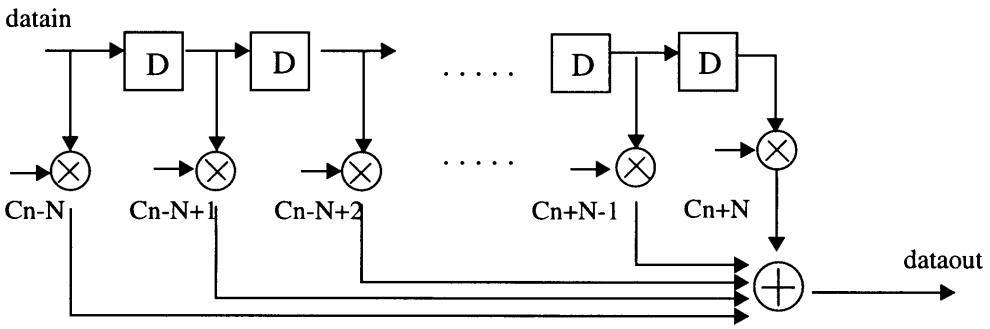


Figure 3.4: A FIR filter implemented using tapped delay line with multipliers and adders.

Control logic

The complexity of address generation logic and datapath depend on the algorithms used and the overall architecture. In addition, logic for coefficients generation is necessary if the filter coefficients are not position-independent, that is, the filter coefficients for each output pixel change across the image (e.g., interpolation), or, the phase of the filter coefficients change over period (e.g. polyphase filter). For linear filters, the coefficients can be generated by logic in real time. As the order of the filter increases, it may be simpler to store the filter coefficients in a lookup table, but this limits the number of scaling factors the scaler is capable to handle.

Memory storage

The amount of internal memory required by the scaler include the input/output buffer, line store, and coefficients store (necessary if one multiply-accumulate unit is used for multiple functions). The sizes of input/output buffer and line storage required are determined by many factors: the speed/pattern of external memory access cycle and display requirement, vertical scaling factor, and the type/width of vertical filter. To take advantage of the maximum memory bandwidth, SGRAM provides page-mode access with memory burst cycle, each burst contains 64-bytes of data. On the other side, scaled images are outputted during the time *WinAct* is high.

Chapter 4

Implementation of Scaling Algorithm

4.1 Issues in Selecting Scaling Methods

Based on the discussions in the previous chapters, we can evaluate various algorithms in terms of hardware complexity and select the ones suitable for implementation. Because of the different issues involved in enlarging and shrinking images, different algorithms are used. Linear Interpolation is used for image enlargement in both vertical and horizontal directions. Area Weighted Averaging is used for horizontal reduction in the horizontal, and for vertical reduction, a combination of Area Weighted Averaging and Bresenham algorithm is used. This chapter will present the process of the selecting algorithms.

4.1.1 Selection based on ranges of scaling factors

One of the major challenges in this design is that the scaler must handle arbitrary scaling factors. Many algorithms discussed in chapter 2 can be implemented easily for a small set of scaling factors, but the cost for implementing these algorithms rises with increasing the number of scaling factors. In addition, the quality of scaling cannot degrade too much with limited precision of scaling factors. In the example of the anti-aliased resampling, where the scaling factor can be any rational number, a polyphase filter is commonly used. A polyphase filter divides the filter coefficients into different phases and sequences through the phases in order to decrease the number of filter taps needed at any point of time, and to avoid the need for large memory to store the interpolated intermediate data and the high processing speed to filter the intermediate data. However, when the size of the input and output become relatively prime, L and M are large, and the design for the polyphase filter becomes very difficult. In the example of anti-alias interpolation methods, the filter coefficients are recalculated for each output pixel. The values of these coefficients depend on the distance from the position of the original pixel to the new pixels, and the width of the filter is stretched or compressed according to the scaling factor. If a non-linear filter is used, calculating these coefficients in real time would require an expensive ALU. One alternative is

to store the coefficients in a look-up table. However, the size of the table grows as the number of scaling factors increase. Furthermore, higher order filters require higher precision, which requires more computation or a larger coefficient table. Other algorithms, for example the Pre-filtering method with pre-calculated coefficients, only work well for a limited number of scaling factors.

The first task in designing a commercial product is to define the goals one wants to achieve based on the type of consumer market. Since the input images come from MPEG1 (352x240) sequences, MPEG2 (720x480) sequences, or computer games (640x480), we can safely assume that the scaling factor for image enlargement will be at most slightly above 2 for MPEG2 and game data, and slightly above 4 for MPEG1 data, given the resolutions of current monitors. Therefore, scaling methods that improve the image quality above these scaling factor, while adding extra complexity in hardware implementation are not practical. For image reduction, the scaling factor can be fairly arbitrary as the image can be shrunk to an icon size. However, consumers generally do not demand high quality for images that are already very small.

Because of the difference in image enlargement and reduction, one may want to use different algorithms for different ranges of scaling factor. However, it is more efficient to reuse the same hardware function for both ranges of scaling factors.

Image enlargement

A 2-tap Linear interpolation is used for image enlargement in both horizontal and vertical directions. The advantages of this method, based on the previous discussions, are:

- It provides much better quality than pixel replication.
- It works for a large range of scaling factors. The scaling quality does not degrade quickly with precision errors if the scaling factor can not be represented by fixed point binary number.
- It is easy to implement. The coefficients are derived directly from the output pixel position.
- Although scaled image quality is not as good as higher order interpolator, the difference is not noticeable for scaling factors less than two. For most applications, its quality is acceptable.

Image reduction

Area-weighted Averaging is used to for image reduction in both horizontal and vertical directions. The advantages of this method are:

- It provides much improved image quality from pixel dropping.
- It works with a large range of scaling factors. The scaling quality does not degrade quickly with precision errors, if the scaling factor can not be represented by fixed point binary number.
- The coefficients are linear, therefore the same hardware used for enlargement can be reused for reduction.
- It is easy to implement. It assigns the same weight to each pixel, therefore no dynamic normalization is required (as opposed to anti-alias interpolation).

The only disadvantage of this method is that it does not provide as much filter overlap between neighboring pixels in the original image compared some other methods, such as the anti-aliased interpolation. However, for most scaling factors, the introduced artifacts are not noticeable, and for large scaling factors, pictures are sometimes too small to tell the differences.

4.1.2 Vertical vs. horizontal scaling

As discussed in the last chapter, scaling in the vertical direction introduces a different challenge from scaling in the horizontal direction. First, image data in the memory are stored as scan lines and can only be accessed as entire lines. If there is not enough memory bandwidth for data access, the logical solution is to drop an entire scan line. As the result, the methods that can be used for vertical scaling depend on number of lines that can be accessed in a given period. Second, each delay element in the vertical direction corresponds to an expensive line store. Therefore, the algorithms implemented for scaling in vertical direction may be different from the scaling in horizontal direction because of these extra constraints.

According to equation 3.2, to access one scan line out of typical computer game data (horizontal width = 640 pixels, 24 bit/pixel), given a 1280×1024 display with approximately 80kHz horizontal line frequency ($\delta T \approx 12.5$ MB/sec), the bandwidth required will

be 230 Mbytes/sec. Since the peak memory bandwidth is 800 Mbytes/sec, in theory the scaler could access 3 new input lines for each output line. In reality, however, the memory bus rarely reaches its peak bandwidth, and the scaler competes for the bandwidth with other graphics functions. That limits the scaler to operating on at most two additional input lines for each output line.

The challenge in scaling as a result of this bandwidth constraint occurs in image reduction, where we want to preserve as much of the source information as possible in order to avoid aliasing. Most anti-aliasing algorithms require the processing of all pixels in the original image. Because the size of the output image is smaller than the input image (decreased sampling rate), but the display horizontal frequency stays constant for a given display type, the bandwidth for processing the entire image increases with the vertical scaling factor. For vertical scaling beyond a 2:1 reduction, these methods would require a minimum of three additional input lines for each output line, and therefore are not feasible. Nevertheless, the anti-aliasing function is implemented for small vertical reduction since the user may still find anti-aliasing to be a necessary feature for a small scaling factor. In image enlargement, a wider vertical filter implies a large overlap between the vertical filter taps because the sampling rate has increased. If there is sufficient line buffer to store the lines that have been previously accessed, the bandwidth limitation does not really apply to the case of image enlargement.

Previously we have chosen linear interpolation as the method for image enlargement and Area Averaging for image reduction. With the limited memory bandwidth, vertical reduction only works for up to 2:1 reduction. For vertical reduction beyond 2:1, Bresenham line dropping algorithm is implemented.

4.2 Algorithm Detail

From the results of the previous discussion and comparison of image quality, two algorithms are chosen as the scaling methods.

4.2.1 Image Enlargement

The pseudo code bellow describes the linear interpolation method for image enlargement.

```
-- cum[i]          = cumulative coefficient
-- c               = coefficient for scaling
-- P[j], P[j+1]    = input pixel pair
-- Q               = output pixel
-- HOldsize        = input width
-- VOldsize        = input height
-- HNewsize        = output width
-- VNewsize        = output height

Initialization:
r          = (Oldsize-1) / (Newsize-1);
cum[0]     = 1;
i          = 0;
j          = 0;
k          = 0;
```

Horizontal Unit

```
Begin loop. (i < HNewsize)
    c <= cum[i];
    Q = c * P[j] + (1 - c) * P[j+1]; /* output pixel ready */

    cum[i+1] = cum[i] - r;
    advance = (cum[i+1] < 0);
    if (advance)   /* goto next pixel pair */
    {
       cum[i+1] = cum[i+1] + 1;
       j = j + 1;
    }

    i = i + 1;
end loop
```

Vertical Unit

```
Begin loop (i < VNewsize)

    c <= cum[i];

    Begin loop (k < HOldsize)
```

```
        Q[k] = c * P[j*HOldsize+ k] + (1 - c) * P[(j+1)*HOldsize+k];
        k = k + 1;
    end loop

    cum[i+1] = cum[i] - r;
    advance = (cum[i+1] < 0);
    if (advance)   /* goto next line pair */
    {
       cum[i+1] = cum[i+1] + 1;
       j = j + 1;
    }

    i = i + 1;
end loop
```

For horizontal enlargement, output pixels are produced at the rate of one pixel per cycle, whereas input pixels are consumed at a slower rate depending on the scale factor. The parameter *cum* keeps track of the output pixel positions and is decremented by *r* at each cycle. Filter coefficients are derived from truncating *cum*. If *cum* is less than zero, the control signal *advance* signals to request a new input pixel, and *cum* is incremented by one. Similarly, for vertical enlargement, output lines are produced for every line iteration, while input lines are consumed at a slower rate. The parameter *cum* keeps track of the output line positions and is decremented by *r* at the end of each scan line. If *cum* is less than zero, the control signal *advance* signals to increment the line address for a new input line, and *cum* is incremented by one.

4.2.2 Image Reduction

The pseudo-code below describes the Area Weighted Averaging method for image reduction.

- Q_{part} = Partial pixel value (for horizontal direction)
- $Q_{part[k]}$ = K_{th} partial pixel value (for vertical direction)

Initialization:
r = Newsize/Oldsize;

```
cum[0] = r;
i = 0;
j = 0;
j = 0;
Q_part = 0;
```

Horizontal Unit

```
Begin loop (i < HNewsize)
    c <= cum[j];
    Q_part  = Q_part + c * P[j] + (1-c) * P[j+1];

    cum[j+1] = cum[j] + r;
    ready   = (cum[j+1] >= 1);

    if (ready)  /* output pixel ready */
    {
      cum[j+1] = cum[j+1] - 1;
      Q[i] = Q_part;
      Q_part = 0;
      i = i + 1;
    }
    else
    {
      Q_part = Q_part - P[j+1];
    }

    j = j + 1;

end loop
```

Vertical Unit

```
Begin loop (i < VNewsize)
    c <= cum[j];
    cum[j+1] = cum[j] + r;
    ready   = (cum[j+1] >= 1);
    if (
    Begin loop (k < HOldsize)
```

```
    Q_part[k] = Q_part + c * P[j*HOldsize+k] + (1-c) * P[(j+1)*HOldsize+k];
    if (ready) {
      Q = Q_part[k]; /* output pixel ready */
      Q_part[k] = 0;
    }
    else Q_part[k] = Q_part[k] - P[(j+1)*O_h+k];
    k = k+1;
  end loop

  if (ready)
  {
    cum[j+1] = cum[j+1] - 1;
    i = i + 1;
  }
  j = j + 1;

end loop
```

For horizontal reduction, input pixels are consumed at the rate of one pixel per cycle, whereas output pixels are produced at a slower rate depending on the scale factor. The parameter *cum*, in this case, is incremented by r at each cycle to keep track of the input pixel position. Filter coefficients are produced by truncating *cum*. If *cum* is greater than one, the current output pixel (the value in Q_{part}) is completed; the parameter *ready* signals to indicate the output is available, and *cum* is decremented by one. If *cum* is less than one, the output pixel is not ready yet, and the value Q_{part} is subtracted by the current pixel value. This algorithm uses the same expression, $Q = c \times P_1 + (1 - c) \times P_2$, as is used in the algorithm enlargement, allowing the same multiply-accumulate unit to be used for both modes of operation. When written as $Q = c \times (P_1 - P_2) + P_2$, this equation only uses one multiplier. Figure 4.1 shows an example of image reduction by weighted averaging. The equation for Area Weighted Averaging can be written as follows:

$$Q[i] = c * P[j] + r * P[j+1] + r * P[j+2] + (1-2*r-c)*P[j+3]$$

$$= c * P[j] + (1-c) * P[j+1] \quad (1)$$
$$+ (-1) * P[j+1]$$
$$+ (r+c) * P[j+1] + (1-r-c) * P[j+2] \quad (2)$$
$$+ (-1) * P[j+2]$$
$$+ (2*r+c) * P[j+2] + (1-2*r-c)*P[j+3] \quad (3)$$

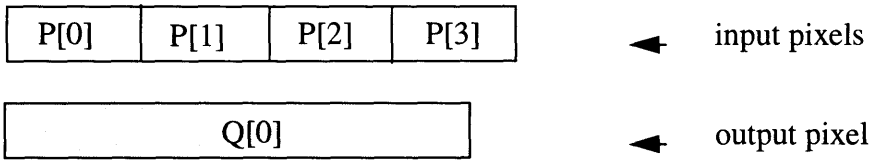


Figure 4.1: Area Weighted Averaging

At the end of the third iteration, the parameter *cum* is greater than 1 and the output pixel is ready. The same analogy can also be applied for vertical reduction.

4.3 Correction for Precision Error

Given the requirement for performing the scaling in real time, as well as the limitations on the complexity and cost of the scaler, it is imperative that the calculations performed by the module use integer or fixed-point arithmetic. Because the algorithm is based on real number arithmetic, precision errors are inevitable when performing the calculation. Using K-bit fixed-point representation of the incrementing factor *r*, the error ε would be less than 2^{-K}. Since *r* is added (or subtracted) to *cum* for each iteration, as shown in the pseudo code, the error in *cum* will accumulate proportionally with the number of pixels handled. That error can have two effects on the algorithm: pixels will be outputted (or inputted for scaling up) at the wrong cycles, and ultimately, the total number of pixels output will be different than the actual number desired. The impact of first effect on output quality can be minimized, as it will be shown later. The second effect, however, can interrupt the synchronization of the pipeline.

Let's determine the exact problems the error causes. Let r be the exact coefficient equal to N/M, where N and M are *Newsize* and *Oldsize* image reduction, and the opposite during image enlargement. Also, let r' be the K-bit approximation of r, and let N' be the actual number of pixels outputted (or inputted in the scaling down case). Given that $0 <= r < 1$, it follows that $r' = r + \varepsilon$, where $-(2^{-K}) < \varepsilon < 2^{-K}$. The accumulated error in cum will then be $M\varepsilon$. While this error will definitely be non-zero in the cases where r does not have exact binary representation, the situation can be simplified a lot by selecting K large enough so that $M\varepsilon < 1$ and thus there would be only one more or one less pixel outputted at the end of a scan line. Given that $M_{max} = 1600$, it follows that $2^K > 1600$, and therefore $K > 10$. All further calculations assume that $K > 10$, so that $M\varepsilon < 1$.

Let's look at the magnitude of the error for each of the two possible approximations: when r is rounded down to r', and when r is rounded up to r'. Let's first look at the case when $r' < r$, i.e. r is rounded down in its binary approximation. In that case $r' = r - \varepsilon$, where $0 \le \varepsilon < 2^{-k}$. From the algorithm for image reduction, it follows that a pixel is outputted at step i only if $\lfloor ir \rfloor > \lfloor (i-1)r \rfloor$. It is easy to see, therefore, that the number of pixels outputted by step i is $\lfloor ir \rfloor$. Thus, when using exact numbers, the total number of pixels outputted is always N:

$$\lfloor Mr \rfloor = \left\lfloor M\frac{N}{M} \right\rfloor = \lfloor N \rfloor = N$$

Notice that the above implies that there is always a pixel outputted at step i=M, since:

$$\lfloor (M-1)r \rfloor = \lfloor N - r \rfloor = N - 1$$

$$\lfloor Mr \rfloor = N > N - 1 = \lfloor (M-1)r \rfloor$$

When the rounded down approximation of r is used, however, the number of pixels outputted would be:

$$\lfloor Mr' \rfloor = \lfloor M(r-\varepsilon) \rfloor = \lfloor N - M\varepsilon \rfloor = N - 1$$

given that $M \times \varepsilon < M \times 2^{-k} < 1$

Thus, when using $r' < r$, there will always be one less pixel outputted. Given that in the ideal case there is always a pixel produced at the last input pixel, one solution of this

problem is to force the scaler to always output a pixel when i=M, if we can be sure that the missing pixel always occur at the last input pixel. This last output pixel will compensate for the missing pixel, and ensure that the output image is of the correct size. The condition that the scaler would not output a pixel when i=M implies that:

$$\lfloor Mr' \rfloor = \lfloor (M-1)r' \rfloor$$

Therefore,

$$\lfloor Mr - M\varepsilon \rfloor = \lfloor Mr - M\varepsilon - r + \varepsilon \rfloor$$

$$\lceil M\varepsilon \rceil = \lceil M\varepsilon + r - \varepsilon \rceil$$

$$M\varepsilon + r - \varepsilon \leq 1$$

$$r \leq 1 - (M-1) \times 2^{-K} \tag{4.1}$$

Now consider rounding r up to r'. In that case, $r' = r + \varepsilon$, where $0 \leq \varepsilon < 2^{-k}$. The number of pixels outputted under these conditions would be:

$$\lfloor Mr' \rfloor = \lfloor N + M\varepsilon \rfloor = N$$

Thus, when $r' > r$, the pixels outputted will always be the correct number. One problem that can occur in this case, however, is that no pixel may be outputted when i=M, which implies the last pixel will be outputted prematurely and thus the last input pixel would be 'ignored'. In order to ensure that the last pixel is outputted when i=M, the following condition must be true:

$$\lfloor Mr' \rfloor > \lfloor (M-1)r' \rfloor$$

Thus,

$$N > \lfloor N + M\varepsilon - r - \varepsilon \rfloor$$
$$M\varepsilon - r - \varepsilon < 0$$

$$r > (M-1) \times 2^{-K} \tag{4.2}$$

From this analysis it follows that both rounding down and rounding up can cause precision problems, unless certain conditions are satisfied. The precision problems can be avoided, however, if the algorithm uses rounding down in some situations, and rounding up in others. If K is increased sufficiently to guarantee that $(M-1)\times 2^{-K} < \frac{1}{2}$, then equations 4.1 and 4.2 can be written as:

$$r \le \frac{1}{2} < 1 - (M-1)\times 2^{-K} \text{ for rounding down} \tag{4.3}$$

and

$$r > \frac{1}{2} > (M-1)\times 2^{-K} \text{ for rounding up} \tag{4.4}$$

Therefore, given $K \ge 12$, the scaler can calculate r' by rounding down r if $r \le \frac{1}{2}$, and by rounding up r if $r > \frac{1}{2}$. This approach will avoid all of the precision error problems discussed above.

Chapter 5

Hardware Implementation

5.1 Design Methodology

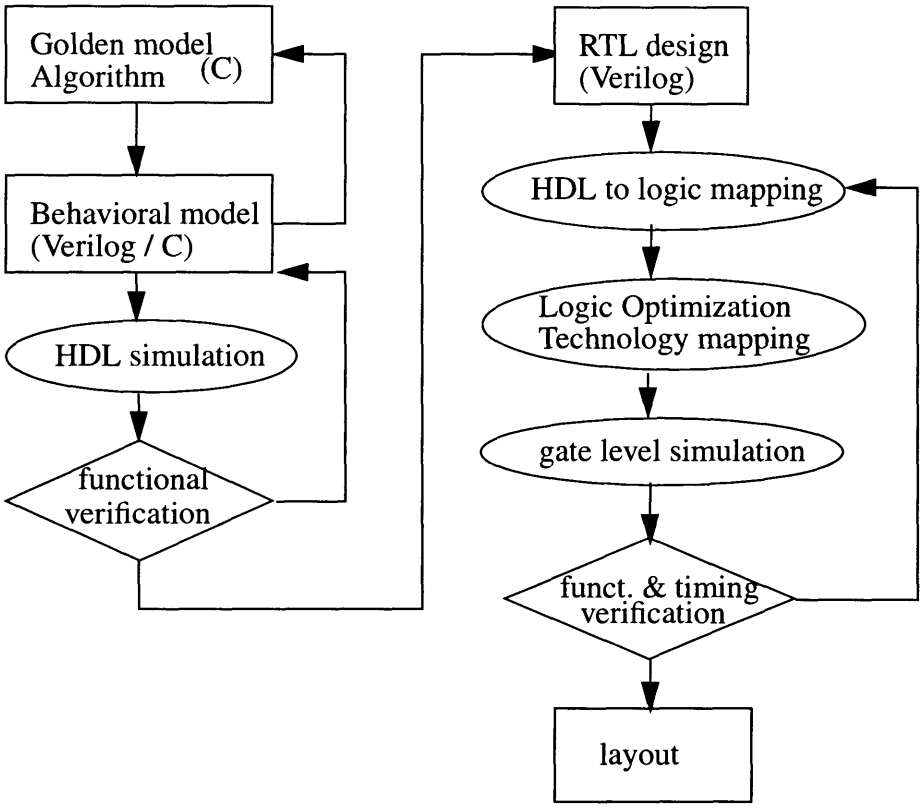


Figure 5.1: Design Methodology

Figure 5.1 shows the methodology used for designing the video/graphics scaler. The scaling algorithm is first modeled in C using the equations described in chapter 2. The C model helps to identify the limitations of the algorithm in terms images quality and to determine the necessary precision of the filter coefficients. It also serves as the golden model of the scaler and is used for validating the correctness of the next steps of the process. The next stage is the behavioral modeling of the scaler using C or Verilog. This model describes the pipeline of the video/graphics scaler hardware. It is used to verify the feasibility of the algorithm implementation in hardware and to identify the dependencies

between different stages of the pipeline. The interfaces between the system modules, synchronous or asynchronous, are also specified at this stage. The behavioral model is used to explore some important issues in the actual hardware design, such as the existence of time critical sections, the scheduling of the control signals, and the resource allocation. The next step in the design process is the RTL (Register Transfer Level) design, which describes the design in terms of registers and operations. The RTL description is synthesized and translated into logic equations. It is important to make the distinction between modeling at the behavioral level and the RTL level. The benefit of behavioral modeling is the ability to architecturally verify a design at a macro (system) level or micro (modular) level before investing the time into producing RTL code. An additional benefit is that the simulation of behavioral code is much faster than low-level simulation. This design goes through the process of logic optimization, gets converted to a gate level design, and finally gets mapped onto the selected technology, and converted to a gate level design. It is important that at each step the design is verified and compared with the previous step to ensure the correct implementation.

This top-down design methodology helps to identify issues regarding algorithmic limitation, architectural constraint and performance estimation during the early stages of the design phase, and avoids the costly process of implementing individual modules before their functionality is verified.

5.2 Video/Graphics Scaler

Figure 5.2 is an abstract representation of the scaler. The five main blocks of the scaler system are the Input Unit, the Vertical Scaling Unit, the Intermediate Asynchronous Buffer, the Horizontal Scaling Unit, and the Output Unit. The Input Unit is responsible for accessing data from the external memory and parsing the information into individual component channel used by the rest of the scaler. The Vertical and Horizontal Scaling Units perform data resampling in their corresponding directions. The Output Unit provides pixel data in a continuous stream at the request of the display. Because of the different clock frequencies of the display and the rest of the system, the scaler utilizes two different synchronization clocks; the first one is the system clock (*SCLK*), which operates at 100MHz, and

the second one is the display pixel clock (*PCLK*), which operates at a range of frequencies depending on the display specifications. The partitioning of the modules in terms of clock frequency is as follows: the Input Unit and the Vertical Scaling Unit operate at the system clock, while the Horizontal Scaling Unit and the Output Unit operate at the display pixel clock. Communication across the clock boundary is achieved through the Intermediate Asynchronous Buffer. The dashed line in the figure indicates the clock boundary.

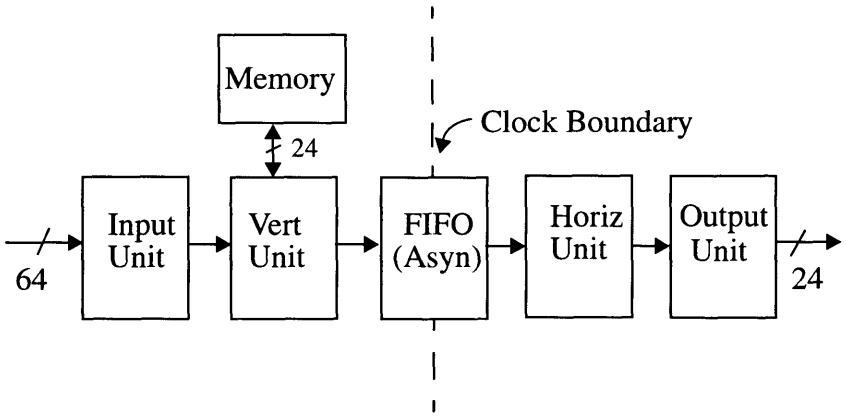


Figure 5.2: Abstract representation of video/graphics scaler

This thesis focuses mainly on the modeling and simulation of the scaling core, including the Vertical Scaling Unit, the Horizontal Scaling Unit, and the Intermediate Asynchronous Buffer. The Verilog code for the modules can be found in Appendix B. The Input and Output Units are not included in the scope of the thesis. Nevertheless, some knowledge about the functionality of these modules is required for the implementation of the scaling core.

5.2.1 Input Unit

The Input Unit shown in figure 5.3 consists of three major components: the external memory address control, input FIFO, and a pixel format unit. The output of the Input Unit consists of two separate data pipelines. Each of the data pipelines contains three component channels. The data in each channel is represented by 8-bit, fixed-point, unsigned numbers. The second pipeline is necessary to meet the high throughput requirement during vertical reduction. As discussed in the previous section, due to the memory bus contention, the vertical scaling is restricted so that at most two additional scan lines are accessed from the

memory for each output line. In this application, the maximum input line size is 720 pixels (MPEG data). To process all 1440 pixels during the display horizontal time (δT_h) which is given by the specification of the monitor, the required speed often exceeds the system clock of 100 MHz. The two pipelines provide pixel data from two separate scan lines simultaneously and align the pixels in the vertical direction.

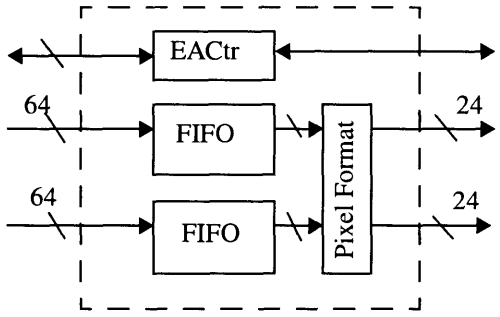


Figure 5.3: Input Unit

External Address Control

External Address Control communicates with the main graphics memory controller. The major responsibilities of the module include calculating the correct line offset in the current image frame and making sure the input FIFO does not overflow. The Bresenham algorithm is implemented to calculate the line offset when memory bandwidth can not meet the demand of the scaler. We assume this condition will occur when the vertical scaling factor is less than 1/2. When two input lines are accessed simultaneously, the address control needs to signal the memory controller to alternate the delivery of the pixel data between these two lines.

Pixel Format Unit

The Pixel Format Unit parses data into three 8-bit component channels, each processed by one of the three component pipelines. The scaler supports several basic data types. The decoded MPEG video data are in YUV color space and in either 4:2:0 or 4:2:2 format. In YUV 4:2:0 format, chrominance data are subsampled by 2 in both horizontal and vertical direction, and the three data components are stored separately in the graphics memory. In YUV 4:2:2 format, chrominance is subsampled in the horizontal direction. The luminance

and chrominance data alternates such that every 32-bit word contains two luminance data and one chrominance data (figure 5.4). The Pixel Format Unit removes the subsampling of YUV data, and adds an offset of 128 to the chrominance data to make it an unsigned number, which simplifies the design of the ALU. Since both algorithms used in scaling are linear, this offset will not change the result of the algorithm and is removed after scaling.

Figure 5.4: YUV Data format

Graphics data are 16 bit or 24 bit, in RGB color space. 16 bit RGB data are presented in either 5:6:5 format, where green components are assigned one extra bit, or 5:5:5 format, where the extra bit is used as a tag. The Pixel Format Unit replicates the lower bits to expand the data to full 8 bit numbers. 24-bit RGB data are presented in a 32-bit word, where the first byte is used for color information such as chroma key values.

Input Buffer

The input FIFO serves as a temporary storage for raw data from the graphics memory. It is vital that the FIFO buffer never becomes empty and always contains enough pixels to satisfy the real-time demands of the display. It is also important that the buffer never overflows and loses display data, thus the level of the FIFOs must be constantly monitored to determine whether more display data must be accessed. There are several other issues which influence the design of the buffer as well: the pattern of memory access, and the format in which the data is stored in the graphics memory. To take advantage of the peak memory bandwidth, burst mode is used as much as possible in data access. The different color components may be stored in different memory blocks (e.g. YUV 4:2:0 format) and thus may need to be accessed during separate cycles. In those cases, each burst of data consists of only one component. The same problem occurs when two scan lines are accessed simultaneously and read in a alternate fashion (one burst cycle from each line). Since pixels are processed one at a time, a single FIFO is not sufficient for the variety of

operations. The choice between having multiple FIFOs and having a single RAM will depend on the complexity of the logic for addressing the pixels.

5.2.2 Vertical Scaling Unit

The Vertical Scaling Unit shown in figure 5.5 consists of a control module, a line buffer, and an arithmetic unit (VALU). The vertical scaler performs scaling at the system clock frequency (SCLK). The two input data paths come from the output of the pixel format unit. Each data path carries a 24-bit pixel, consisting of three 8-bit component channels. The memory block serves as a line buffer which stores the previously fetched pixel data for calculation of the next output line, thus minimizing main memory access. The MUX selects the data stream to be stored in the line buffer. The possible selections are each of the two data paths, as well as the output of the line buffer itself. The arithmetic unit performs data computation.

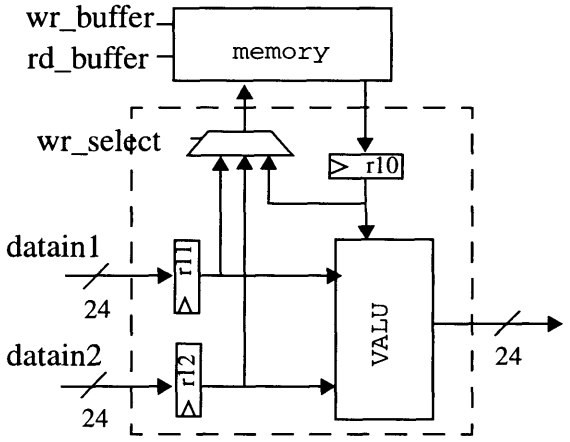


Figure 5.5: Vertical Scaling Unit (control module is not shown in the figure for simplicity)

Vertical Control Logic

The controls for the Vertical and Horizontal Unit use a basic **scaling engine** which is responsible for calculating the position of the output lines, as well as producing the linear coefficients. The incrementing factor *r*, as calculated in the previous chapter, is represented as a 12-bit, fixed point number. Since the calculation of *r* involves division, and is only calculated once for each scaling factor, the most hardware-efficient solution is to do the calculation in software and load it into a register. The internal variable *cum* is the accumulating term, which has 13-bit of precision. The most significant bit detects over/under

flow, which is used to determine the output signals ADV and READY. The coefficient C1 is a 6-bit number produced by truncating cum. This precision is limited by the speed of the one-cycle **multiply-accumulate** unit. Pipelining the **multiply-accumulate** was considered but not implemented, because of the extra complexity and the fact that for the linear scaling methods, 6 bits of coefficient precision provides sufficient output quality.

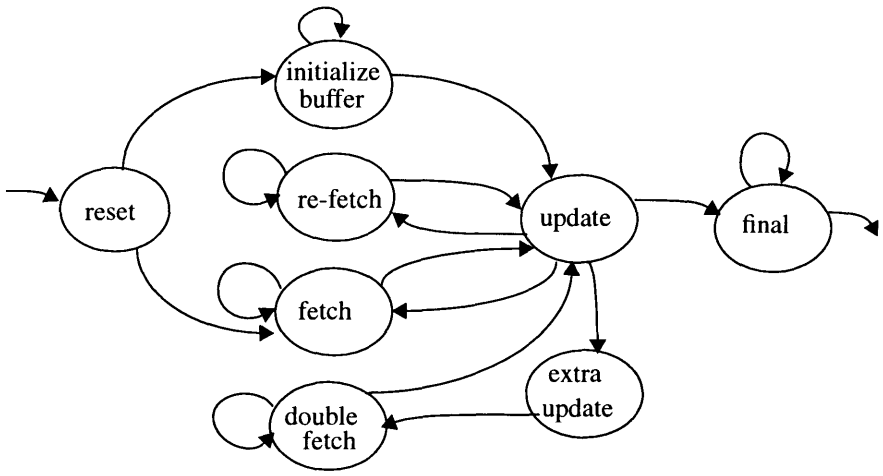


Figure 5.6: Vertical Unit state diagram

The Vertical Control generates signals that regulate data flow in the Vertical Unit. As shown in the state diagram in figure 5.6, there are total of 8 states. In the 'reset' state, the registers are initialized with the correct values. The next five states correspond to the different modes of operation of the vertical scaling. The 'initialize buffer' state is a special case which occurs in the initialization of the first input scan line during vertical reduction. Since the first output line is a weighted average of the first and second input lines, the first input line is stored in the line buffer, and no output is available. The 're-fetch' state occurs during vertical enlargement. If the position of the next output line stays in between the same two input lines, the previously stored scan line in the line buffer is written back into the memory, while the current input line in the first pipeline is discarded. The 'fetch' state occurs both during vertical enlargement and reduction. The current input line in the first pipeline is stored in the line buffer for calculating the next output line. The 'double fetch' state occurs in vertical reduction when two input lines are accessed simultaneously. The current input line in the second pipeline is stored in the line buffer for the calculation of

the next output line. The 'final' state occurs when the last output line is being computed. It is treated as a special case as the input and output signals are modified according to the error correction mechanism discussed in 4.3. The last two states are transitional states. The 'update' state occurs at the end of each scan line. During an 'update' the accumulate term *cum* in the scaling engine is incremented, and the state for the next scan line is determined. The 'extra update' state occurs only during vertical reduction. If the *ready* signal from scaling engine is not active after the 'update' state, an output line is not completed, and an additional input scan line needs is needed. In that case the scaling engine is updated once again and produces the second filter coefficient for the 'double fetch' state. An input pixel counter is used to keep track of the position in the current scan line and to generate the end of scan line signal (*scan_end)*. Two interrupt signals cause the pipeline to stall: the first comes from the empty signal of the input FIFO indicating data are not available, the second one comes from the full signal of the FIFO buffer between the Vertical Unit and the Horizontal Unit. During stall, the entire pipeline is disabled and no changes of state occur. Table 5.1 lists the important output signals for each state. Table 5.2 summarizes the conditions for transitions among states.

Table 5.1: Output signals for Vertical Control Module

	wr_buffer	rd_buffer	wr_select	double fetch	stall_engine
reset	0	0	xx	0	1
init buffer	1	0	01	0	1
re-fetch	1	1	00	0	1
fetch	1	1	01	0	1
double fetch	1	1	10	1	1
update	0	0	xx	0	0
extra update	0	0	xx	0	0
final	0	1	xx	0	1

Table 5.2: Vertical Control State Transition Table

Current State	Next State	Inputs
reset	init buffer	$\overline{\text{Vmode}}$
reset	fetch	Vmode
init buffer	init buffer	$\overline{\text{scan\_end}}$
init buffer	update	scan_end
re-fetch	re-fetch	$\overline{\text{scan\_end}}$
re-fetch	update	scan_end
fetch	fetch	$\overline{\text{scan\_end}}$
fetch	update	scan_end
double fetch	double fetch	$\overline{\text{scan\_end}}$
double fetch	update	scan_end
update	final	last_line
update	fetch	$\overline{\text{last\_line}}$ & ((ADV & Vmode) | (READY &$\overline{\text{Vmode}}$))
update	re-fetch	$\overline{\text{last\_line}}$ & $\overline{\text{ADV}}$ & Vmode
update	extra update	$\overline{\text{last\_line}}$ & $\overline{\text{READY}}$ & $\overline{\text{Vmode}}$
extra update	extra update	TRUE
final	final	$\overline{\text{scan\_end}}$
final	reset	scan_end

Vertical Arithmetic Unit (VALU)

The vertical arithmetic unit shown in figure 5.7 performs calculations for each component channel. The output of the multiply-and-accumulate unit (MAU) are represented with 14 bit precision (8 bit for the data and 6 bit for the coefficients). Truncation is performed to keep the 8 most significant bits. When image enlargement is performed, only the top multiply-accumulate unit is used. When image reduction is performed, both multiply-accumulate units are used. The output of the vertical unit is selected by the multiplexer. Synchronization of the pixels in the line buffer and the two pipelines is very important because pixels in the same column have to arrive the Vertical Unit at the same cycle in

order to perform the correct averaging.

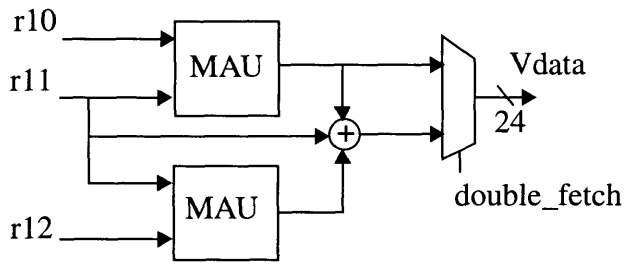


Figure 5.7: Vertical Arithmetic Unit

Line Storage

The data components are packed into to 24-bit word and stored in a memory array of size 720x24 (the maximum line size is 720). The memory consists of four memory banks of sizes 16x24, 64x24, 128x24, and 512x24. Each memory bank has two ports; one for read access and the other for write access.

5.2.3 Horizontal Scaling Unit

The Horizontal Scaling Unit consists of a control module and a arithmetic module. It is simpler than the Vertical Scaling Unit, as it needs to keep only a single pixel of state, as opposed to an entire scan line. The Horizontal Unit operates at the Display clock frequency (PCLK). Again, each data path carries a 24-bit pixel, consisting of three 8-bit component channels.

Horizontal Control Logic

The scaling engine used by the horizontal unit is the same as discussed in the vertical unit. The only difference is that the engine is updated after each cycle, instead of at the end of a scan line.

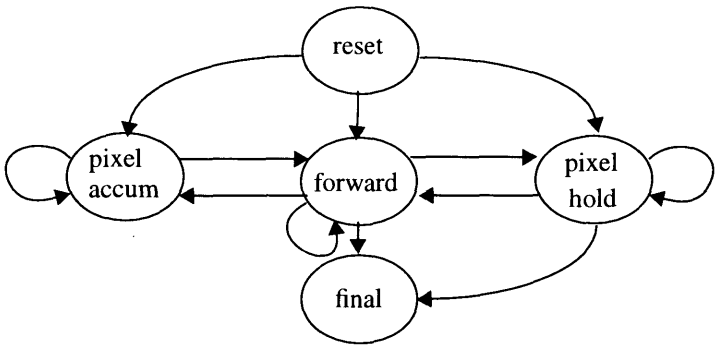


Figure 5.8: Horizontal Unit state diagram

The Horizontal Control generates signals that regulate data flow in the Horizontal Unit. As shown in the state diagram in figure 5.8, there are 5 states in the Horizontal Control. In 'reset' state, the registers are initialized to the correct values. The 'pixel accum' state occurs only in horizontal reduction. In this state, the horizontal scaler consumes a new input pixel, while no output pixel is produced. Partial pixel data is stored in a register. The 'pixel hold' state occurs only in horizontal enlargement. In this state, the horizontal produces an output pixel but does not consume a new input pixel because the output pixel mapped to the same two input pixels. The 'forward' state occurs in both horizontal enlargement and reduction. The horizontal scaler consumes a new input pixel and produces an output pixel. In this state, the register used to store partial pixel data (reduction only) is reset to zero (clr = 1). The 'final' state indicates the last pixel for a scan line. Again, signals are modified according to the error correction mechanism discussed in 4.3. Two signals causes the horizontal pipeline to stall: the first is the EMPTY signal from the intermediate buffer, which indicates no input data are available, the other is the FULL signal from the output FIFO which indicates that the display has not requested output pixel data. Table 5.3 summarizes the conditions for transitions among states.

Table 5.3: Horizontal Control State Transition Table

Current State	Next State	Inputs
All except final	pixel accum	$\overline{\text{ready}}$ & advance
All except final	forward	ready & advance
All except Final	pixel hold	ready & $\overline{\text{advance}}$
All except reset	final	last_pixel
final	reset	true

Horizontal Arithmetic Unit (HALU)

The Horizontal Unit shown in figure 5.9 performs the calculation for each component channel. One multiply-add unit is used for both horizontal enlargement and reduction. 14 bits of precision are maintained at the output of the multiplier (8 bits of data and 6 bits of coefficients), and truncation is performed for the final data. Pixels values are clipped at zero for under flow. The register Q_{part} is used to stored the partial pixel data during horizontal reduction.

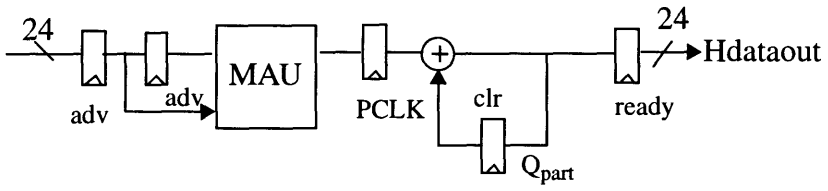


Figure 5.9: Horizontal Arithmetic Unit

5.2.4 Intermediate Asynchronous FIFO

The FIFO serves as an intermediate buffer between the Vertical Scaling Unit and the Horizontal Scaling Unit. The FIFO operates asynchronously. Data are written into memory on low-to high transition at the system clock (*SCLK*) input and is read out on a low-to-high transition at the display pixel clock (*PCLK*) input. When the memory is full, *SCLK* signals have no effect. When the memory is empty, *PCLK* signals have no effect. The status of the FIFO is monitored by the FULL, EMPTY, and FULL-1, EMPTY-1 output flags.

5.2.5 Output Unit

The external signal *WINACT* is given to the Output Unit, indicating that the display is requesting pixel data. While *WINACT* is inactive, the scaler continues to operate until the output FIFO is full. The FIFO then signals the horizontal unit, causing it to stall. The horizontal unit, in return, causes the vertical unit to stall in a few cycles as well by letting the intermediate buffer fill up. When *WINACT* becomes active, the display starts reading pixels from the output FIFO. The newly available space in the FIFO activates the scaler and makes it calculate the remaining pixels of the scan line. Given that the display reads an entire scan line in a single continuous interval, it is imperative that this FIFO is never empty when *WINACT* is active.

Output FIFO

The output FIFO provides buffer between the scaler and the display, permitting slower-than-real-time processing when the demand for real-time scaling exceeds input bandwidth and processing speed. The buffer allows the scaler to use the entire horizontal period to compute one output line. The minimum size of the FIFO required by the scaler can be approximated through the following calculation.

The Vertical Unit, which operates at the frequency of the system clock (*SCLK*), produces pixels at maximum rate of F_s = 100M (pixels/sec). The Horizontal Unit, which operates at the frequency of the display pixel clock (*PCLK*), produces pixels at a rate of F_d = 30 - 220M (pixels/sec). For $F_d > F_s$, the limiting factor of the system is the speed of the Vertical Unit, therefore pixels are produced at $S \times F_s$ M (pixels/sec), where S is the horizontal scaling factor. Since the active window time (*WINACT*) is $N \times S / F_d$ (sec), where N is the horizontal width of the original image, the number of pixels that can be produced in real time is $\frac{N \times S}{F_d} \times S \times F_s$. Therefore the number of pixels that need to be pre-calculated and stored in the buffer is:

$$L = N \times S - N \times S^2 \times \frac{F_s}{F_d} = N \times S \times \left(1 - S \times \frac{F_s}{F_d}\right) \quad . \tag{5.1}$$

According to this equation, no line buffer is required when $S = 0$ and $S = F_d / F_s$. By setting the derivative with respect to S zero, we find that the maximum required buffer size is

$$L_{max} = \frac{N}{4} \times \frac{F_d}{F_s} \tag{5.2}$$

For $F_d < F_s$, the limiting factor is the Horizontal Unit. Because input pixels are processed at one pixel per cycle, pixel preprocessing is needed for image reduction. Following the same steps as the previous case, the number of pixels that can be produced in real time is $\frac{N \times S}{F_d} \times S \times F_d = N \times S^2$, so the number of pixels that need to be preprocessed is:

$$L = N \times S \times (1 - S). \tag{5.3}$$

Again, by taking the derivative, we find the size required for the output buffer is

$$L_{max} = \frac{N}{4} \tag{5.4}$$

Given that $N = 720$ is the largest line size by MPEG2 standard, and the maximum display frequency is 220MHz, the maximum required output buffer size is

$$L_{max} = \frac{720}{4} \times \frac{220MHz}{100MHz} = 396 pixels \tag{5.5}$$

This buffer size is the minimum requirement for the scaler, given that the system always operates at its optimum speed without interruption. Since the number of bytes in the memory arrays are powers of two, a buffer size of 512 is used in this implementation.

5.3 Testing

In contrast to the top-down design methodology described in the first section, a bottom-up strategy is used for testing. Each individual module is simulated and tested before being integrated into the system. The advantage of this approach is that it narrows the scope of the problems and simplifies the debugging process.

Two major modules are tested separately: the Horizontal Module (Hmodule), the Vertical Module (Vmodule). Since commercial memory arrays are used for the line storage and are not yet available and this stage of design, a simple memory module, using the same protocol as a FIFO, is written as a stub for the purposes of simulation and testing. To reduce the amount of time for simulation, small image dimensions are used initially. Test drivers are written which generate input data with known output results and provide them at the correct cycle time. The following test cases are used in the simulation. Note that r is the incrementing factor, and r' = Round(oldsize-1 / newsize-1) for image enlargement, and r = Round(newsize/oldsize) for image reduction.

1. Enlargement and reduction. Test for large r (r approaches 1) and small r (r approaches 0).

2. Pipeline stall. Test if the correct state is presumed and whether pixel data are lost after the stalling.

The next step is to integrate the Vertical Unit and the Horizontal Unit using a intermediate buffer. Again, a small FIFO module is written as a stub in place of commercial memory array for simulation and testing. A single clock (*PCLK* = *SCKL*) is used to simplify the simulation, although two separate clocks can be used with the same protocol. Similar test cases, as described above, are used for testing the modules. In addition, different combination of scaling mode are tested: 1. vertical and horizontal enlargement, 2. vertical and horizontal reduction, 3. vertical enlargement and horizontal reduction, 4. vertical reduction and horizontal enlargement.

Finally, real images are used as input for the scaler and tested with various scaling factors. The output of the scaler are saved in a file and compared with the output of the C model. Figure 5.10 describes this procedure.

The Verilog code for the modules and the test drivers can be found in Appendix B. The Horizontal and Vertical Unit are in **Hmodule** and **Vmodule** respectively. The pseudo line buffer and intermediate FIFO are in **lbuffer** and **buffer**. The modules are integrated into one module called **scaler**.

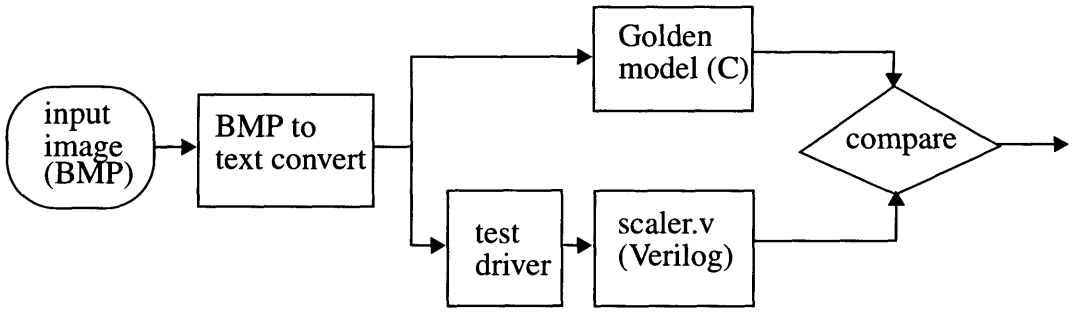


Figure 5.10: Flow chart for testing the scaler

Chapter 6

Conclusion

This thesis investigated the advantages and disadvantages of various methods for scaling video and graphics images. By comparing the quality of the scaled images and analyzing the issues involved in implementing scaling algorithms in real-time hardware, algorithms that compromised between hardware complexity and quality of the image were selected. Based on the discussions developed in the early chapters, a real-time hardware implementation of the selected algorithms was proposed and simulated.

As discussed in the first chapter, high performance real-time image scaling can be achieved by specialized hardware. However, the constraints imposed by hardware issues, such as video subsystem architecture, performance requirement, and cost of implementation, limit the complexity for feasible algorithms. Furthermore, due to the different criteria involved in image enlargement and reduction, as well as the different frequency contents of various types of source images, it is often undesirable to apply one scaling algorithm to all situations. This thesis explored this idea by applying Linear Interpolation to image enlargement, and a combination of Area Weighted Averaging and Bresenham algorithm to image reduction. In addition, the amount of hardware cost was reduced by taking advantage of the similarities between the algorithms and calculating the minimum storage requirements.

In the design of the hardware scaler, memory bandwidth was the main bottleneck for performance, especially in the case of vertical reduction. For small scaling factors below a certain threshold it becomes impossible to deliver all pixels of the source image to the scaler in real-time. Similar problems are frequently encountered in digital signal processor architectures for many other applications and often present interesting topics for research.

Until real-time performance can be realized in software, hardware scaling remain to be an important topic for study. Many more ideas can be explored in either the architectural or the algorithmic level. For example, video system architectures need to be improved to satisfy the demands of the digital signal processing algorithms. On the other hand, the digital signal processing algorithms can be improved both to minimize the amount of compu-

tation required, as well as to address other practical issues, such as human visual responses to the resulting artifacts.

Appendix A

C code for Scaling Algorithms

```
/*************************************************************************

   Last Modified: Erika Chuang 8/6/96
   This program rescales the source image into the desired image size.
   It works under both UNIX and DOS environment. The source and resulting
   images are in 24-bit RGB color format, and BMP image file format.
   *************************************************************************/
#include <stdio.h> #include <string.h> #include <malloc.h> #include <math.h>
#include "test.h"

main(int argc, char *argv[]) {

   FILE *fs;
   FILE *fd;
   unsigned short header[27];
   int c;
   int xsize, ysize;
   char filein[32];
   char fileout[32];
   double (*filter_type)()=filter;
   double filter_width;
   Image *dst, *src, *tmp;
   char flag[10], x_filter[24], y_filter[24];
   /* Request input image file name */
   printf("Input file: ");
   scanf("%s", filein);
   fs = fopen(filein,"rb");
   if (fs == NULL){
      printf("couldn't open file: %s for reading\n", filein);
      exit(1);
   }
   /* Request output image file name */
   printf("Output file: ");
   scanf("%s", fileout);
   fd = fopen(fileout,"wb");
   if (fd == NULL){
      printf("couldn't open file: %s for writing\n", fileout);
      exit(1);
   }
   fread(header, 2, 27, fs);
   printf("1.UNIX,  2.DOS:  ");
   scanf("%d", &c);
   printf("Horizontal size: ");
   scanf("%d", &xsize);
   printf("Vertical size: ");
   scanf("%d", &ysize);
   /* Load the image from the BMP file.  The byte orders are different
   in UNIX and in DOS. */
   if (c == 1) {
      printf("call load_image_UNIX\n");
      src = load_image_UNIX(fs, xsize, ysize, header);
   }
   else if (c == 2) {
```

```
   printf("call load_image_DOS\n");
   src = load_image_DOS(fs, xsize, ysize, header);
}
/* Selecting a filter */
printf("filter options: c=cubic, t=triangle, b=bell, B=B-spline\n");
printf("  r=replication, l2=bilinear(2), l4=bilinear(4)\n");
printf("                 M=Mitchell, L=Lanczos3, s=truncate sinc\n");
printf("                 a=AreaAVG\n");
printf("Select horizontal filter option: ");
scanf("%s", x_filter);
printf("Select vertical filter option: ");
scanf("%s", y_filter);
switch(*x_filter) {
case 'c': filter_type=filter; filter_width=filter_support; break;
case 't': filter_type=triangle_filter; filter_width=triangle_support;
   printf("triangle\n");break;
case 'b': filter_type=bell_filter; filter_width=bell_support; break;
case 'B': filter_type=B_spline_filter; filter_width=B_spline_support; break;
case 'M': filter_type=Mitchell_filter; filter_width=Mitchell_support; break;
case 'L': filter_type=Lanczos3_filter; filter_width=Lanczos3_support; break;
case 's': filter_type=sinc; filter_width=sinc_support; break;
}
switch(*y_filter) {
case 'c': filter_type=filter; filter_width=filter_support; break;
case 't': filter_type=triangle_filter; filter_width=triangle_support;
   printf("triangle\n");break;
case 'b': filter_type=bell_filter; filter_width=bell_support; break;
case 'B': filter_type=B_spline_filter; filter_width=B_spline_support; break;
case 'M': filter_type=Mitchell_filter; filter_width=Mitchell_support; break;
case 'L': filter_type=Lanczos3_filter; filter_width=Lanczos3_support; break;
case 's': filter_type=sinc; filter_width=sinc_support; break;
}
/* call the scaling rountine */
tmp = new_image(xsize, src->ysize);
if (strcmp(x_filter, "r") == 0) {
   replicate(tmp, src, "x");
}  else if (strcmp(x_filter, "l2") == 0) {
   bilinear2(tmp, src, "x");
} else if (strcmp(x_filter, "l4") == 0) {
   bilinear4(tmp, src, "x");
} else if (strcmp(x_filter, "a") == 0) {
   AreaAverage(tmp, src, "x");
} else {
   fzoom(tmp, src, filter_type, filter_width, "x");
}
free_image(src);
dst = new_image(xsize, ysize);
if (strcmp(y_filter, "r") == 0) {
   replicate(dst, tmp, "y");
}  else if (strcmp(y_filter, "l2") == 0) {
   bilinear2(dst, tmp, "y");
} else if (strcmp(y_filter, "l4") == 0) {
   bilinear4(dst, tmp, "y");
} else if (strcmp(y_filter, "a") == 0) {
   AreaAverage(dst, tmp, "y");
} else {
   fzoom(dst, tmp, filter_type, filter_width, "y");
}
save_image(dst, header, fd, xsize, ysize);
```

```
   free_image(dst);
   free_image(tmp);
   fclose(fs);
   fclose(fd);
   exit(0);
}

/****************************************

   Scaling algorithms:
   1. fzoom(): anti-aliased interpolation
   2. replicate(): DDA and Bresenham algorithm
   3. bilinear2(): 2 taps
   4. bilinear4(): 4 taps
   5. AreaAverage

   Last modified: Erika 4/16/97
   Add area-weigted averagin into the code
   ****************************************/
#include <stdio.h> #include <math.h> #include <malloc.h> #include "test.h"

double filter(double t);

double box_filter(double t);

double triangle_filter(double t);

double bell_filter(double t);

double B_spline_filter(double t);

double sinc(double t);

double Lanczos3_filter(double t);

double Mitchell_filter(double t);

void fzoom(Image *dst, Image *src,

double (*filter)(),

double fwidth, char *flag);

void replicate(Image *dst, Image *src, char *flag);

void bilinear2(Image *dst, Image *src, char *flag);

void bilinear4(Image *dst, Image *src, char *flag);

void AreaAverage(Image *dst, Image *src, char *flag);

void area_weight(Image *dst, Image *src, char *flag,

int xsize, int ysize, int newxsize, int newysize,

unsigned short r);

/** filter function **/

/* cubic function */ double filter(double t) {

   /* f(t) = 2|t|^3 - 3|t|^2 + 1, -1 <= t <= 1  */
   if(t < 0.0) t = -t;
```

```
   if(t < 1.0) return((2.0 * t - 3.0) * t * t + 1.0);
   return (0.0);
}

/* replcation */ double box_filter(double t) {

   if((t > -0.5) && (t <= 0.5)) return(1.0);
   return(0.0);
}

/* linear interpolation */ double triangle_filter(double t) {

   if(t < 0.0) t = -t;
   if(t < 1.0) return(1.0 - t);
   return(0.0);
}

double bell_filter(double t)   /* box (*) box (*) box */ {

   if(t < 0) t = -t;
   if(t < .5) return(.75 - (t * t));
   if(t < 1.5) {
      t = (t - 1.5);
      return(.5 * (t * t));
   }
   return(0.0);
}

double B_spline_filter(double t)    /* box (*) box (*) box (*) box */ {

   double tt;
   if (t < 0) t = -t;
   if (t < 1) {
      tt = t * t;
      return((.5 * tt * t) - tt + (2.0 / 3.0));
   }
   else if (t < 2) {
      t = 2 - t;
      return((1.0 / 6.0) * (t * t * t));
   }
   return(0.0);
}

double sinc(double x) {

   x *= 3.14159;
   if(x != 0) return(sin(x) / x);
   return(1.0);
}

double Lanczos3_filter(double t) {

   if(t < 0) t = -t;
   if(t < 3.0) return(sinc(t) * sinc(t/3.0));
   return(0.0);
   }


double Mitchell_filter(double t) {

   double tt;
```

```
   tt = t * t;
   if(t < 0) t = -t;
   if(t < 1.0) {
      t = (((12.0 - 9.0 * B - 6.0 * C) * (t *tt))
      + ((-18.0 + 12.0 * B + 6.0 * C) * tt)
      + (6.0 - 2 * B));
      return(t/6.0);
   } else if(t < 2.0) {
      t = (((-1.0 * B - 6.0 * C) * (t * tt))
      + ((6.0 * B + 30.0 * C) * tt)
      + ((-12.0 * B - 48.0 * C) * t)
      + (8.0 * B + 24 * C));
      return(t/6.0);
   }
   return(0.0);
}

typedef struct {

   int pixel;  /* which pixel in the line */
   double weight;/* weight contributed by the pixel */
} CONTRIB;

typedef struct {

   int n;/* number of pixel contribution */
   CONTRIB *p;/* pointer to a list of CONTRIB pixels */
} CLIST;

CLIST *contrib;

/** scaling routine **/

void fzoom(Image *dst, Image *src, double (*filter)(), double fwidth, char *flag)
{

   int i,j,k;
   int pixel;
   double xscale, yscale;     /* scaling factors */
   double center, left, right;/* filter cal variable */
   double weight, width, fscale;/* weighing factor, width of filter */
   double red, green, blue, sum;
   Pixel *data;
   Line *row, *col;
   /* create an intermediate image to hold horizontal scaling */
   xscale = (double) dst->xsize / (double) src->xsize;
   yscale = (double) dst->ysize / (double) src->ysize;
   if (strcmp(flag, "x") == 0) {
      printf("horizontal fzoom .....\n");
      /* Calculate filter contributions to a row */
      contrib = (CLIST *) calloc(dst->xsize, sizeof(CLIST));
      if (xscale < 1.0) {
      width = fwidth / xscale;/* width of filter */
      fscale = 1.0/xscale;/* normalize sum of weight to one */
      for (i=0; i < dst->xsize; i++) {
            contrib[i].n = 0;
            contrib[i].p = (CONTRIB *) calloc((((int) 2*(int)width + 1))*3,
            sizeof(CONTRIB));
            center = (double) i / xscale;/* center of filter position */
            left = ceil(center - width);/* leftmost pixel */
```

```
      right = floor(center + width);/* rightmost pixel */
      sum = 0.0;/* sum of weights */
      for (j = left; j <= right; j++) {
      weight = center - (double) j;
      weight = (*filter)(weight/fscale);
      if (j < 0) {/* out of boundary */
      pixel = -j;
      } else if (j >= src->xsize) {
      pixel = src->xsize - (j - src->xsize ) - 1;
      } else {
      pixel = j;
      }
      k = contrib[i].n++;
      sum += weight;
      contrib[i].p[k].weight = weight;
      contrib[i].p[k].pixel = pixel;
      }
      /* normalize the sum of weights to 1 if greater than one */
      /* if (sum > 1.0) */
      for (j=0;j<=k;j++)
      contrib[i].p[j].weight /= sum;
      }
}
else {
printf("fwidth %f\n", fwidth);
printf("first left %f\n", ceil(0.0-fwidth));
for (i=0; i < dst->xsize; i++) {
      contrib[i].n = 0;
      contrib[i].p = (CONTRIB *) calloc(((int) (2*(int)fwidth + 1)*3),
      sizeof(CONTRIB));
      center = (double) i / xscale;
      left = ceil(center - fwidth);
      right = floor(center + fwidth);
      for (j = left; j <= right; j++) {
      weight = center - (double) j;
      weight = (*filter)(weight);
      if (j < 0) {/* out of boundary */
      pixel = -j;
      } else if (j > src->xsize) {
      pixel = src->xsize - (j - src->xsize ) - 1;
      } else {
      pixel = j;
      }
      k = contrib[i].n++;
      contrib[i].p[k].weight = weight;
      contrib[i].p[k].pixel = pixel;
      }
      }
}
/* calculate the output from input pixel weights */
row = new_line(src->xsize);
data = malloc(sizeof(Pixel));
for (k=0;k <dst->ysize; k++) {
get_row(row, src, k);
for (i=0; i<dst->xsize; i++) {
      red = 0.0;
      green = 0.0;
      blue = 0.0;
      for (j=0; j<contrib[i].n; j++) {
```

```
        red += row->red[contrib[i].p[j].pixel] * contrib[i].p[j].weight;
        green += row->green[contrib[i].p[j].pixel] * contrib[i].p[j].weight;
        blue += row->blue[contrib[i].p[j].pixel] * contrib[i].p[j].weight;
        }
        if (red < 0) red = 0;
        if (green < 0) green = 0;
        if (blue < 0) blue = 0;
        if (red > 255) red = 255;
        if (green > 255) green = 255;
        if (blue > 255) blue = 255;
        data->red = (unsigned char) red;
        data->green = (unsigned char) green;
        data->blue = (unsigned char) blue;
        put_image(dst, i, k, data);
        }
   }
   free(data);
   /* deallocate memory space for horizontal filter weight */
   free_line(row);
   for (i=0; i<dst->xsize; i++) {
   free(contrib[i].p);
   }
   free(contrib);
}
if (strcmp(flag, "y") == 0) {
printf("vertical fzoom .....\n");
/* Calculate filter contributions to a column */
contrib = (CLIST *) calloc(dst->ysize, sizeof(CLIST));
if (yscale < 1.0) {
   width = fwidth / yscale;/* width of filter */
   fscale = 1.0/yscale;/* normalize sum of weight to one */
   for (i=0; i < dst->ysize; i++) {
   contrib[i].n = 0;
   contrib[i].p = (CONTRIB *) calloc((((int) 2*(int)width + 1))*3,
   sizeof(CONTRIB));
   center = (double) i / yscale;/* center of filter position */
   left = ceil(center - width);/* leftmost pixel */
   right = floor(center + width);/* rightmost pixel */
   sum = 0.0;/* sum of weights */
   for (j = left; j <= right; j++) {
        weight = center - (double) j;
        weight = (*filter)(weight/fscale);
        if (j < 0) {/* out of boundary */
        pixel = -j;
        } else if (j >= src->ysize) {
        pixel = src->ysize - (j - src->ysize ) - 1;
        } else {
        pixel = j;
        }
        k = contrib[i].n++;
        contrib[i].p[k].weight = weight;
        contrib[i].p[k].pixel = pixel;
        sum += weight;
        }
        /* normalize sum of weights to 1 if greater */
        /* if (sum > 1.0) */
        for (j=0;j<=k;j++)
        contrib[i].p[j].weight /= sum;
   }
```

```
    } else {
       for (i=0; i < dst->ysize; i++) {
       contrib[i].n = 0;
       contrib[i].p = (CONTRIB *) calloc((((int) 2*(int)fwidth + 1))*3,
       sizeof(CONTRIB));
       center = (double) i / yscale;
       left = ceil(center - fwidth);
       right = floor(center + fwidth);
       for (j = left; j <= right; j++) {
            weight = center - (double) j;
            weight = (*filter)(weight);
            if (j < 0) {/* out of boundary */
            pixel = -j;
            } else if (j > src->ysize) {
            pixel = src->ysize - (j - src->ysize ) - 1;
            } else {
            pixel = j;
            }
            k = contrib[i].n++;
            contrib[i].p[k].weight = weight;
            contrib[i].p[k].pixel = pixel;
            }
       }
    }
       col = new_line(src->ysize);
       data = malloc(sizeof(Pixel));
       for (k=0;k <dst->xsize; k++) {
       get_column(col, src, k);
       for (i=0; i<dst->ysize; i++) {
            red = 0.0;
            green = 0.0;
            blue = 0.0;
            for (j=0; j<contrib[i].n; j++) {
            red+=col->red[contrib[i].p[j].pixel] * contrib[i].p[j].weight;
            green+=col->green[contrib[i].p[j].pixel] * contrib[i].p[j].weight;
            blue+=col->blue[contrib[i].p[j].pixel] * contrib[i].p[j].weight;
            }
            if (red < 0) red = 0;
            if (green < 0) green = 0;
            if (blue < 0) blue = 0;
            if (red > 255) red = 255;
            if (green > 255) green = 255;
            if (blue > 255) blue = 255;
            data->red = (unsigned char) red;
            data->green = (unsigned char) green;
            data->blue = (unsigned char) blue;
            put_image(dst, k, i, data);
            }
       }
       free(data);
       /* deallocate memory space for vertical filter weight */
       free_line(col);
       for (i=0; i<dst->ysize; i++) {
       free(contrib[i].p);
       }
       free(contrib);
    }
}
```

```
/* replicates or decimates pixels based on Bresenham line drawing algorithms*/
void replicate(Image *dst, Image *src, char *flag) {

   int i,j,k;
   int dx, dy, e, dx2, y;
   Pixel *data;
   Line *row, *col;
   data = malloc(sizeof(Pixel));
   /* first do horizontal scaling */
   if (strcmp(flag, "x") == 0) {
   printf("horizontal replicating ....\n");
   dx = dst->xsize;
   dy = src->xsize;
   dx2 = dx*2;
   e = 2*dy - dx;
   dy = 2*dy;
   row = new_line(src->xsize);
   for (k=0;k<src->ysize;k++) {
      get_row(row, src, k);
      y = 0;
      for (i=0;i<dx;i++) {
      data->red = row->red[y];
      data->green = row->green[y];
      data->blue = row->blue[y];
      put_image(dst,i,k,data);
      while (e>0) {
            y++;
            e-=dx2;
            }
            e+=dy;
      }
   }
   free_line(row);
   }
   /* Vertical scaling */
   if (strcmp(flag, "y") == 0) {
   printf("vertical replicating ....\n");
   dx = dst->ysize;
   dy = src->ysize;
   dx2 = dx*2;
   e = 2*dy - dx;
   dy = 2*dy;
   col = new_line(src->ysize);
   for (k=0;k<src->xsize;k++) {
      get_column(col, src, k);
      y = 0;
      for (i=0;i<dx;i++) {
      data->red = col->red[y];
      data->green = col->green[y];
      data->blue = col->blue[y];
      put_image(dst,k,i,data);
      while (e>0) {
            y++;
            e-=dx2;
            }
            e+=dy;
      }
   }
   }
```

```
   free(data);
  }


/* bilinear2 averages 2 neighboring pixels in the input images

    some aliasing effect should be significant at small scale down factor ***/
void bilinear2(Image *dst, Image *src, char *flag) {
   int i,j,k;
   Line *row, *col;
   Pixel *data;
   double w1, w2, center, xscale, yscale, xdelta, ydelta, align;
   xscale = (double) dst->xsize/(double) src->xsize;
   yscale = (double) dst->ysize/(double) src->ysize;
   center = 0;
   xdelta = 1/xscale;
   ydelta = 1/yscale;
   printf("bilinear interpolation..\n");
   row = new_line(src->xsize);
   data = malloc(sizeof(Pixel));
   if (strcmp(flag, "x") == 0) {
   printf("horizontal linear interpolating .....\n");
   for (k=0;k<dst->ysize;k++) {
      get_row(row, src, k);
      j = 0;
      align = 0;
      center = 0;
      for (i=0;i<dst->xsize;i++) {
      if (align == 0) {
            data->red = row->red[j];
            data->green = row->green[j];
            data->blue = row->blue[j];
            put_image(dst,i,k,data);
            } else {
            w2 = align;
            w1 = 1 - w2;
            data->red =
            (unsigned char) (w1 * row->red[j] + w2 * row->red[j+1]);
            data->green =
            (unsigned char) (w1 * row->green[j] + w2 * row->green[j+1]);
            data->blue =
            (unsigned char) (w1 * row->blue[j] + w2 * row->blue[j+1]);
            put_image(dst,i,k,data);
            }
            center+=xdelta;
            if (center >= (j+1)) {
            j= floor(center);
            align = center - j;
            }
            else if (j > src->xsize) {
            j = src->xsize - 1;
            align = 0;
            }
            else {align = center - j;}
      }
   }
   free_line(row);
   }
   if (strcmp(flag, "y") == 0) {
```

```
   printf("vertical linear interpolating .....\n");
   col = new_line(src->ysize);
   for (k=0;k<dst->xsize;k++) {
      get_column(col, src, k);
      j = 0;
      align = 0;
      center = 0;
      for (i=0;i<dst->ysize;i++) {
      if (align == 0) {
            data->red = col->red[j];
            data->green = col->green[j];
            data->blue = col->blue[j];
            put_image(dst,k,i,data);
            } else {
            w2 = align;
            w1 = 1 - w2;
            data->red = (unsigned char) (w1 * col->red[j] + w2 * col->red[j+1]);
            data->green = (unsigned char) (w1 * col->green[j] + w2 * col-
>green[j+1]);
            data->blue = (unsigned char) (w1 * col->blue[j] + w2 * col->blue[j+1]);
            put_image(dst,k,i,data);
            }
            center+=ydelta;
            if (center >= (j+1)) {
            j=floor(center);
            align = center - j;
            }
            else if (j > src->ysize) {
            j = src->ysize - 1;
            align = 0;
            }
            else {align = center - j;}
      }
   }
   free_line(col);
   }
   free(data);
}

void bilinear4(Image *dst, Image *src, char *flag) {

   int i,j,k;
   Line *row, *col;
   Pixel *data;
   double w1, w2, w3, w4, center, xscale, yscale, xdelta, ydelta, align;
   xscale = (double) dst->xsize/(double) src->xsize;
   yscale = (double) dst->ysize/(double) src->ysize;
   center = 0;
   xdelta = 1/xscale;
   ydelta = 1/yscale;
   printf("xdelta = %f, ydelta = %f\n", xdelta, ydelta);
   printf("bilinear interpolation..\n");
   row = new_line(src->xsize);
   data = malloc(sizeof(Pixel));
   if (strcmp(flag, "x") == 0) {
   printf("horizontal linear interpolating .....\n");
   for (k=0;k<dst->ysize;k++) {
      get_row(row, src, k);
      j = 0;
```

```
for (k=0;k<dst->xsize;k++) {
   get_column(col, src, k);
   j = 0;
   align = 0;
   center = 0;
   for (i=0;i<dst->ysize;i++) {
   if (align == 0) {
         data->red = col->red[j];
         data->green = col->green[j];
         data->blue = col->blue[j];
         put_image(dst,k,i,data);
         } else {
         w2 = 0.5 * (1 - 0.5 * align);
         w3 = 0.75 - w2;
         w4 = 0.5 * 0.5 * align;
         w1 = 0.25 - w4;
         /* overflow case */
         if (j == 0) {
         data->red = (unsigned char) (w1 * col->red[j+1] + w2 * col->red[j]
         + w3 * col->red[j+1] + w4 * col->red[j+2]);
         data->green =
         (unsigned char) (w1 * col->green[j+1] + w2 * col->green[j]
         + w3 * col->green[j+1] + w4 * col->green[j+2]);
         data->blue = (unsigned char) (w1 * col->blue[j+1] + w2 * col->blue[j]
         + w3 * col->blue[j+1] + w4 * col->blue[j+2]);
         } else if ((j+2) == src->xsize) {
         data->red = (unsigned char) (w1 * col->red[j-1] + w2 * col->red[j]
         + w3 * col->red[j+1] + w4 * col->red[j-1]);
         data->green =
         (unsigned char) (w1 * col->green[j-1] + w2 * col->green[j]
         + w3 * col->green[j+1] + w4 * col->green[j-1]);
         data->blue = (unsigned char) (w1 * col->blue[j-1] + w2 * col->blue[j]
         + w3 * col->blue[j+1] + w4 * col->blue[j-1]);
         } else { /* normal case */
         data->red = (unsigned char) (w1 * col->red[j-1] + w2 * col->red[j]
         + w3 * col->red[j+1] + w4 * col->red[j+2]);
         data->green =
         (unsigned char) (w1 * col->green[j-1] + w2 * col->green[j]
         + w3 * col->green[j+1] + w4 * col->green[j+2]);
         data->blue = (unsigned char) (w1 * col->blue[j-1] + w2 * col->blue[j]
         + w3 * col->blue[j+1] + w4 * col->blue[j+2]);
         }
         put_image(dst,k,i,data);
         }
         center+=ydelta;
         if (center >= (j+1)) {
         j=floor(center);
         align = center - j;
         } else if (j > src->ysize) {
         j = src->ysize - 1;
         align = 0;
         }
         else {align = center - j;}
   }
}
free_line(col);
}
free(data);
```

```
   align = 0;
   center = 0;
   for (i=0;i<dst->xsize;i++) {
   /* for center of filter lies on a input pixel */
   if (align == 0) {
        data->red = row->red[j];
        data->green = row->green[j];
        data->blue = row->blue[j];
        put_image(dst,i,k,data);
        } else {
        w2 = 0.5 * (1 - 0.5 * align);
        w3 = 0.75 - w2;
        w4 = 0.5 * 0.5 * align;
        w1 = 0.25 - w4;
        /* overflow case */
        if (j == 0) {
        data->red = (unsigned char) (w1 * row->red[j+1] + w2 * row->red[j]
        + w3 * row->red[j+1] + w4 * row->red[j+2]);
        data->green =
        (unsigned char)(w1 * row->green[j+1] + w2 * row->green[j]
        + w3 * row->green[j+1] + w4 * row->green[j+2]);
        data->blue = (unsigned char) (w1 * row->blue[j+1] + w2 * row->blue[j]
        + w3 * row->blue[j+1] + w4 * row->blue[j+2]);
        } else if ((j+2) == src->xsize) {
        data->red = (unsigned char) (w1 * row->red[j-1] + w2 * row->red[j]
        + w3 * row->red[j+1] + w4 * row->red[j-1]);
        data->green =
        (unsigned char) (w1 * row->green[j-1] + w2 * row->green[j]
        + w3 * row->green[j+1] + w4 * row->green[j-1]);
        data->blue = (unsigned char) (w1 * row->blue[j-1] + w2 * row->blue[j]
        + w3 * row->blue[j+1] + w4 * row->blue[j-1]);
        } else { /* normal case */
        data->red = (unsigned char) (w1 * row->red[j-1] + w2 * row->red[j]
        + w3 * row->red[j+1] + w4 * row->red[j+2]);
        data->green =
        (unsigned char) (w1 * row->green[j-1] + w2 * row->green[j]
        + w3 * row->green[j+1] + w4 * row->green[j+2]);
        data->blue = (unsigned char) (w1 * row->blue[j-1] + w2 * row->blue[j]
        + w3 * row->blue[j+1] + w4 * row->blue[j+2]);
        }
        put_image(dst,i,k,data);
        }
        center+=xdelta;
        if (center >= (j+1)) {
        j= floor(center);
        align = center - j;
        } else if (j >= src->xsize) {
        j = src->xsize - 1;
        align = 0;
        }
        else {align = center - j;}
   }
}
free_line(row);
}
printf("horizontal scale complete\n");
if (strcmp(flag, "y") == 0) {
printf("vertical linear interpolating .....\n");
col = new_line(src->ysize);
```

```
}

void AreaAverage(Image *dst, Image *src, char *flag) {

   int ysize2;
   unsigned short temp, r;
   if (dst->ysize < (0.5 * src->ysize)) {
      ysize2 = dst->ysize * 2;
   } else {
      ysize2 = src->ysize;
   }
   if (strcmp(flag, "x") == 0) {
      temp = (dst->xsize << 13) / src->xsize;
      r = (unsigned short) temp;
      printf("r factor = %d\n", r);
      area_weight(dst, src, "x",
                     src->xsize, src->ysize, dst->xsize, ysize2, r);
   } else if (strcmp(flag, "y") == 0) {
      temp = (dst->ysize << 13) / ysize2;
      r = (unsigned short) temp;
      printf("r factor = %d\n", r);
      area_weight(dst, src, "y",
      dst->xsize, ysize2, dst->xsize, dst->ysize, r);
   }
}

void area_weight(Image *dst, Image *src, char *flag, int xsize, int ysize, int
newxsize, int newysize, unsigned short r) {

   int i,j, k, n, np, nc;
   unsigned long p, pp;
   short cum;
   Pixel *data1, *data2, *Q;
   int partial_sum_red, partial_sum_green, partial_sum_blue;
   Line *row, *col;
   int coeff1, coeff2, temp;
   int dx, dy, dx2, e, y;
   np = 13;/* spatial precision */
   nc = 6;/* coefficient precision */
   p = (unsigned long) (1 << nc);
   pp = (unsigned long) (1 << np);
   /* horizontal scaling: area weighted averaging */
   dx = newysize;
   dy = ysize;
   dx2 = dx*2;
   e = 2*dy - dx;
   dy = 2*dy;
   row = new_line(src->xsize);
   data1 = malloc(sizeof(Pixel));
   data2 = malloc(sizeof(Pixel));
   Q = malloc(sizeof(Pixel));
   if (strcmp(flag, "x") == 0) {
      row = new_line(xsize);
      y = 0;
      for (n=0;n<dx;n++) {
      /* begin horizontal scaling */
      get_row(row, src, y);
      cum = r;
      partial_sum_red = 0;
```

```
partial_sum_red = 0;
partial_sum_red = 0;
k = 0;
coeff1 = (int) cum >> (np - nc);
coeff2 = p - coeff1;
for (i=0;i<xsize - 1;i++) {
      data1->red = row->red[i];
      data2->red = row->red[i+1];
      data1->green = row->green[i];
      data2->green = row->green[i+1];
      data1->blue = row->blue[i];
      data2->blue = row->blue[i+1];
      partial_sum_red +=
      (int)((coeff1 * data1->red + coeff2 * data2->red) >> nc);
      partial_sum_green +=
      (int)((coeff1 * data1->green + coeff2 * data2->green) >> nc);
      partial_sum_blue +=
      (int)((coeff1 * data1->blue + coeff2 * data2->blue) >> nc);
      if (cum >= pp) {
      if ((partial_sum_red > 255) || (partial_sum_red < 0))
      {
      partial_sum_red = 0;
      }
      if ((partial_sum_green > 255) || (partial_sum_green < 0))
      {
      partial_sum_green = 0;
      }
      if ((partial_sum_blue > 255) || (partial_sum_blue < 0))
      {
      partial_sum_blue = 0;
      }
      Q->red = (unsigned char) partial_sum_red;
      Q->green = (unsigned char) partial_sum_green;
      Q->blue = (unsigned char) partial_sum_blue;
      put_image(dst, k, n, Q);
      /* dst[n*newxsize + k] = (unsigned char) partial_sum; */
      k++;
      partial_sum_red = 0;
      partial_sum_green = 0;
      partial_sum_blue = 0;
      cum -= pp;
      coeff1 = (int) cum >> (np - nc);
      coeff2 = p - coeff1;
      } else {
      partial_sum_red -= data2->red;
      partial_sum_green -= data2->green;
      partial_sum_blue -= data2->blue;
      coeff1 += (r >> (np - nc));
      coeff2 = p - coeff1;
      }
      cum += r;
      }
      /* end horizontal scaling, calculating the address of next line */
      while (e>0) {
      y++;
      e-=dx2;
      }
      e+=dy;
}
```

```
      free_line(row);
   }
   /* horizontal scaling: area weighted averaging */
   if (strcmp(flag, "y") == 0) {
      col = new_line(ysize);
      for (j=0;j<xsize;j++) {
      get_column(col, src, j);
      k = 0;
      cum = (short) r;                /* initialize the sequence */
      partial_sum_red = 0;           /* initialize the partial data accum */
      partial_sum_green = 0;          /* initialize the partial data accum */
      partial_sum_blue = 0;           /* initialize the partial data accum */
      coeff1 = (int) cum >> (np - nc);
      coeff2 = p - coeff1;
      for (i=0;i<ysize;i++) {
            data1->red = col->red[i];
            data2->red = col->red[i+1];
            data1->green = col->green[i];
            data2->green = col->green[i+1];
            data1->blue = col->blue[i];
            data2->blue = col->blue[i+1];
            partial_sum_red +=
            (int) ((coeff1 * data1->red + coeff2 * data2->red) >> nc);
            partial_sum_green +=
            (int) ((coeff1 * data1->green + coeff2 * data2->green) >> nc);
            partial_sum_blue +=
            (int) ((coeff1 * data1->blue + coeff2 * data2->blue) >> nc);
            if (cum >= pp) {
            if ((partial_sum_red > 255) || (partial_sum_red < 0))
            {
            partial_sum_red = 0;
            }
            if ((partial_sum_green > 255) || (partial_sum_green < 0))
            {
            partial_sum_green = 0;
            }
            if ((partial_sum_blue > 255) || (partial_sum_blue < 0))
            {
            partial_sum_blue = 0;
            }
            Q->red = (unsigned char) partial_sum_red;
            Q->green = (unsigned char) partial_sum_green;
            Q->blue = (unsigned char) partial_sum_blue;
            put_image(dst,j,k,Q);
            /*dst[k*newxsize + j] = (unsigned char) partial_sum;*/
            cum -= pp;
            coeff1 = (int) cum >> (np - nc);
            coeff2 = p - coeff1;
            k++;
            partial_sum_red = 0;
            partial_sum_green = 0;
            partial_sum_blue = 0;
            } else {
            partial_sum_red -= data2->red;
            partial_sum_green -= data2->green;
            partial_sum_blue -= data2->blue;
            /*if (partial_sum > 255) printf("overflow\tpartial_sum = %d\n",
partial_sum );*/
            coeff1 += (r >> (np - nc));
```

```
            coeff2 = p - coeff1;
            }
            cum += r;
            }
    }
    free_line(col);
} }
```

Appendix B

Verilog Code for Video and Graphics Scaler

B.1 Modules

B.1.1 Horizontal Unit

```
`timescale 1ns/100ps `define COEFF_PR 6  // number of bit of precision for the
filter coefficients `define SPACE_PR 13 // number of bit of precision for the
increment factor

// ******  Horizontal Module  ******

module Hmodule(clk, reset, stall, Hstall, mode, data_val, datain, Hdataout, Hdata,
r, NEWXSIZE, OLDXSIZE, last_line, Vreset);

    /* ------------------- system inputs -------------------------- */
    input       clk;
    input       reset;
    input       mode;
    input [10:0]       NEWXSIZE, OLDXSIZE;
    input [`SPACE_PR:0] r;
    /* ------------------- functional input ------------------------ */
    input       stall;     // stall signal from the output buffer
    input       data_val;  // input data available
    input [23:0]       datain;    // input data
    input       Vreset;    // signal from Vertical scaler for the
    // completion of a frame
    output       Hstall;    // 0: requesting for an new input data
    wire       Hstall;
    output       Hdataout;  // signal when output is ready
    reg       Hdataout;
    output [23:0]       Hdata;     // output data
    reg [23:0]       Hdata;
    // Define registers
    reg [7:0]       r20, g20, b20;
    reg [7:0]       r21, g21, b21;
    reg [7:0]       rout, gout, bout;
    reg [9:0]       rtemp, gtemp, btemp;
    reg [9:0]       rpart1, gpart1, bpart1;
    reg [9:0]       rpart2, gpart2, bpart2;
    reg [10:0]       HOldCounter, HNewCounter;
    reg [`COEFF_PR-1:0] hc_delay;
    reg       Hreset;
    output       last_line;
    reg       last_line;
    // signals for the engine
    wire [1:0] state;
    wire adv, ready;
    wire [`COEFF_PR-1:0] coeff;
    wire active;
    // Instantiate engine
    engine g(clk,
    Hreset,
```

```
mode,
r,
~active,
ready,
adv,
coeff,
state);
// define reset
always @ (posedge reset) begin
Hdataout = 0;
Hdata = 0;
Hreset = 1;
HOldCounter = 0;
HNewCounter = 0;
last_line = 0;
rpart1 <= 0;
gpart1 <= 0;
bpart1 <= 0;
end
/* ------------------------------------------------------
 * stop getting pixels from the input when
 * 1. input data not available
 * 2. reset
 * 3. advance signal is low during scaling up
 *    or reach the end of the line
 * 4. output buffer is full
 * ------------------------------------------------------ */
assign Hstall = (~data_val & ~last_line) || Hreset ||
(mode && (!adv || HOldCounter == OLDXSIZE)) ||
stall;
always @ (data_val) if (!data_val && !reset && Vreset) last_line = 1;
// last_line goes high for a few cycles at the end of the frame.
always @ (posedge clk) if (last_line && Hreset) last_line = 0;
// the signal drops to signal the completion of the frame.
assign active = (~reset && data_val) || (~Hreset & last_line);
always @ (posedge clk) begin
hc_delay <= coeff;
if (!Hstall) begin
r21 = r20;
g21 = g20;
b21 = b20;
r20 = datain[23:16];
g20 = datain[15:8];
b20 = datain[7:0];
end
if (active)
if (Hreset) begin
Hreset = 0;
Hdataout = 0;
end
else begin
if (mode) begin
Hreset = (HNewCounter+1 == NEWXSIZE);
Hdataout = 1;
if (!state || Hreset)
Hdata = {r20, g20, b20};
else begin
rout = mul_acc(r21, r20, coeff);
gout = mul_acc(g21, g20, coeff);
```

```
bout = mul_acc(b21, b20, coeff);
Hdata = {rout, gout, bout};
end
if (Hreset)
HNewCounter <= 0;
else
HNewCounter <= HNewCounter + 1;
if (Hreset)
HOldCounter <= 0;
else if (!Hstall)
HOldCounter <= HOldCounter + 1;
else
HOldCounter <= HOldCounter;
end
else begin
Hreset = (HOldCounter+1 == OLDXSIZE);
if (Hreset)
HOldCounter <= 0;
else
HOldCounter <= HOldCounter + 1;
if (Hreset)
HNewCounter <= 0;
else if (Hdataout)
HNewCounter <= HNewCounter + 1;
Hdataout = ready || Hreset;
// always output the last pixel even not ready
// these are immediate
rpart2 = rpart1 + mul_acc(r21, r20, hc_delay);
gpart2 = gpart1 + mul_acc(g21, g20, hc_delay);
bpart2 = bpart1 + mul_acc(b21, b20, hc_delay);
rout = clamp_value(rpart2);
gout = clamp_value(gpart2);
bout = clamp_value(bpart2);
Hdata = {rout, gout, bout};
// rpart1 are registers (synchronized)
if (Hdataout || state == 0) begin
rpart1 <= 0;
gpart1 <= 0;
bpart1 <= 0;
end
else begin
rpart1 <= rpart2 - r20;
gpart1 <= gpart2 - g20;
bpart1 <= bpart2 - b20;
end
end // else: !if(mode)
end
else
Hdataout = 0;
end
// Function to implement output = c * d1 + (1 - c) * d2
function [9:0]  mul_acc;
input [7:0]    d1, d2;
input [`COEFF_PR-1:0] c1;
reg [`COEFF_PR+7:0]   acc;
begin
acc = c1 * d1 + ((1 << `COEFF_PR) - c1) * d2;
mul_acc = acc[`COEFF_PR+7:`COEFF_PR];
end
```

```
    endfunction
    // clamp the pixel at 0 and 255
    function [7:0] clamp_value;
    input [9:0]  a;
    begin
    if (a[9] == 1)  clamp_value = 0; // negative
    else if (a > 255) clamp_value = 255;
    else clamp_value = a[7:0];
    end
    endfunction
endmodule // Hmodule
```

B.1.2 Vertical Unit

```
`timescale 1ns/100ps `define COEFF_PR 6  // number of bit of precision for the
filter coefficients `define SPACE_PR 13 // number of bit of precision for the
increment factor

/* ---------------------  Vertical Module ---------------------- */

module Vmodule(clk, reset, rd_data1, rd_data2, mode, buffer_full, data_val1,
data_val2, data1, data2, Vdataout, Vdata, r, NEWYSIZE, OLDYSIZE, OLDXSIZE, Vstate,
Vreset, scan_end);

    /* ------------------ System Input --------------------------- */
    input       clk;      // system clock
    input       reset;      // system reset
    // Input from register file
    input       mode;      // 1:scale up 0:scale down
    input [10:0]       NEWYSIZE;              // original vertical size
    input [10:0]       OLDYSIZE;              // new vertical size
    input [10:0]       OLDXSIZE;              // original line size
    input [`SPACE_PR:0] r;                     // incrementing factor
    /* ------------------ Functional Inputs ----------------------- */
    // Input from output buffer
    input       buffer_full;           // output buffer full
    // Input from input buffer
    input       data_val1, data_val2;  // input data valid
    input [23:0]       data1, data2;          // input data
    /* ------------------ Funtional Outputs ----------------------- */
    // output for output buffer
    output       Vdataout;      // Output data valid
    output [23:0]       Vdata;      // Output data
    reg       Vdataout;
    reg [23:0]       Vdata;
    // output for input buffer
    output       rd_data1, rd_data2;    // requesting new data
    wire       rd_data1, rd_data2;
    // output to Horizontal Unit
    output       Vreset;                // signal end of a frame
    reg       Vreset;
    // output to address control (currently used by test_scaler)
    output       scan_end;      // signal end of a scan line
    output [2:0]       Vstate;                // state of the vertical scaling
    reg       scan_end;
    reg [2:0]       Vstate;
```

```
/* ------------------ Define variables ------------------------- */
reg [7:0]       r11, g11, b11, r12, g12, b12;
reg [7:0]       rout, gout, bout;
reg [9:0]       rpart1, gpart1, bpart1;
reg [9:0]       rpart2, gpart2, bpart2;
// internal signals and registers
wire       pause;                 // stalling signal for the
// vertical scaler
reg [10:0]       PCounter;      // pixel counter, counts for
// one scan line
reg [10:0]       VNewCounter;       // line counter
reg [10:0]       VOldCounter;       // line counter
reg [`COEFF_PR-1:0] coeff_delay;      // coefficients store
reg [`COEFF_PR-1:0] coeff_double;      // coefficients store
reg [`COEFF_PR-1:0] coeff_temp;      // coefficients store
reg       Vout, Vout1, Vstall;
reg       scan_end_dly;
reg       double_fetch;
reg       double_fetch_dly;
reg       adv_state, adv_state_dly;
wire       state_reset;
wire       data_val;
// Module instantiation: signals to/from the engine
wireready, adv;
wire [`COEFF_PR-1:0] coeff;
wire [1:0]state;
wirestall_engine;
engine g(clk, state_reset, mode,  r, stall_engine, ready, adv,
coeff, state);
// Module instantiation: signals to/from line buffer
reg wr, rd;       // write and read line buffer
wire [23:0] bufferin;
wire [23:0] bufferout;
wire full, empty;
lbuffer l(reset, clk, rd, wr, bufferin, bufferout, full, empty);
// Module instantiation: signals for the multiplexor
reg [1:0] wr_select;
reg [1:0] wr_select1;   // input to mux, write select
mux4by24 m(bufferout, {r11,g11,b11}, {r12,g12,b12},data1,wr_select1,
bufferin);
/* ------------------------------------------------------------
 * stop the vertical scaler when
 * 1. reset
 * 2. output buffer full
 * 3. input data not available
 * 4. transition from one scan line to the next
 * ------------------------------------------------------------ */
assign pause = reset || Vreset || buffer_full || !data_val || scan_end ||
scan_end_dly || (double_fetch & ~double_fetch_dly) ||
(adv_state & ~adv_state_dly);
/* ------------------------------------------------------------
 * reset, vertical scaler is idle
 * ------------------------------------------------------------ */
assign state_reset = reset | Vreset;
always @ (reset) Vreset = reset;
always @ (Vreset) begin
if (Vreset) begin
Vstate = 0;
wr = 0;
```

```
rd = 0;
Vout = 0;
scan_end = 0;
PCounter = 0;
VNewCounter = 0;
VOldCounter = 0;
Vstall = 0;
scan_end_dly = 0;
double_fetch = 0;
double_fetch_dly = 0;
adv_state = 0;
adv_state_dly = 0;
end
end
/* ------------------------------------------------------------
 * define interface with the input
 * ------------------------------------------------------------ */
assign rd_data1 = ~pause & ~Vstall;
assign rd_data2 = ~pause & ~Vstall & (Vstate == 4);
// fetching input data
always @ (posedge clk) begin
if (!pause && !Vstall) begin
r11 <= data1[23:16];
g11 <= data1[15: 8];
b11 <= data1[ 7: 0];
if (Vstate == 4) begin
r12 <= data2[23:16];
g12 <= data2[15: 8];
b12 <= data2[ 7: 0];
end
end
end
// synchronize the two pipeline. during double fetch, wait til data for
// both pipeline are available
assign data_val = ((Vstate != 4) & data_val1) ||
((Vstate == 4) & data_val1 & data_val2);
/* ------------------------------------------------------------
 * Define some control signal
 * ------------------------------------------------------------ */
always @ (pause) begin
if (!pause) begin
case (Vstate)
// first line, save the input, output only if scaling up
3'b000: begin
wr = 1;
rd = 0;
wr_select = 1;
Vout = (mode) ? 1: 0;
end
// not advacing to the next line, save
3'b001: begin
wr = 1;
rd = 1;
wr_select = (Vstall) ? 1 : 0;
Vout = 1;
end
// advancing to the next line. save the data from pipeline 1
3'b010: begin
wr = 1;
```

```
rd = 1;
wr_select = 1;
Vout = 1;
end
// double fetch, save the data from pipeline 2
3'b100: begin
wr = 1;
rd = 1;
wr_select = 2;
Vout = 1;
end
endcase
end
else begin
wr = 0;
rd = 0;
Vout = 0;
// wr_select = x;
end
end
// two cycles delay from input to output data available
always @ (posedge clk) Vout1 <= Vout;
always @ (posedge clk) Vdataout <= Vout1;
// one cycle delay from input to writing the line buffer
always @ (posedge clk) wr_select1 <= wr_select;
/* ------------------------------------------------------------------
 * Transitions at the end of scan line
 * ------------------------------------------------------------------ */
always @ (posedge clk) begin
if (!state_reset && !buffer_full) begin
scan_end <= (PCounter+1 == OLDXSIZE);
// reach the end of scan line
if (scan_end)
PCounter <= 0;
// have not reached the end of scan line, update pixel counter
else if (!pause) PCounter <= PCounter + 1;
end
end
/* ------------------------------------------------------------------
 * Updating engine at the end of scan line.
 * Update twice when
 * 1. At the end of the first scan line when scaling up.
 * purpose: run the engine one state ahead in order to determine
 * whether the scan line is stored in the line buffer or the previous
 * scan that is stored in the line buffer.
 * 2. For double fetch when scaling down, engine is incremented twice.
 * More details from timing diagram
 ------------------------------------------------------------------ */
always @ (posedge clk) begin
scan_end_dly <= scan_end;
double_fetch <= (state == 3) ? 1: 0;
double_fetch_dly <= double_fetch;
adv_state <= mode && (VNewCounter == 1);
adv_state_dly <= adv_state;
end
// signal to un-stall the engine (update engine)
assign stall_engine = ~((scan_end & ~scan_end_dly) ||
(double_fetch & ~double_fetch_dly) ||
(adv_state & ~adv_state_dly));
```

```
always @ (state or posedge scan_end_dly) begin
if (double_fetch) Vstate <= 4;
else Vstate <= state;
end
always @ (scan_end) begin
if (!scan_end  && !state_reset) begin
// scaling up
if (mode) begin
Vreset = (VNewCounter+1 == NEWYSIZE);
if (VNewCounter+2 == NEWYSIZE)
Vstall = (NEWYSIZE == OLDYSIZE) ? 0: 1;
if (Vreset)
VNewCounter <= 0;
else if (!reset) begin
VNewCounter <= VNewCounter + 1;
end
if (Vreset)
VOldCounter <= 0;
else if (adv)
VOldCounter <= VOldCounter + 1;
end
// scaling down
else begin
Vreset = (VOldCounter+1 >= OLDYSIZE);
if (Vreset)
VOldCounter <= 0;
else if (Vstate == 3)
VOldCounter <= VOldCounter + 2;
else if (!reset)
VOldCounter <= VOldCounter + 1;
end
end
end
/* --------------------- coefficients ------------------------ */
always @ (posedge clk) begin
if (scan_end) coeff_temp <= coeff;
else coeff_temp <= coeff_temp;
coeff_delay <= coeff_temp;
end
always @ (posedge double_fetch or adv_state) coeff_double <= coeff;
/* ------------------- pixel calculation ----------------------- */
always @ (posedge clk) begin
if (mode) begin
if (Vstall)
// last line
Vdata <= bufferout;
else if ((Vstate == 0) || (~Vstall &&(NEWYSIZE == OLDYSIZE)))
// first line or last line when scaling factor is 1
Vdata <= {r11, g11, b11};
else if (adv_state) begin
// second line
rout = mul_acc(bufferout[23:16], r11, coeff_double);
gout = mul_acc(bufferout[15:8], g11, coeff_double);
bout = mul_acc(bufferout[7:0], b11, coeff_double);
Vdata <= {rout, gout, bout};
end
else begin
// normal case
rout = mul_acc(bufferout[23:16], r11, coeff_delay);
```

```
    gout = mul_acc(bufferout[15:8], g11, coeff_delay);
    bout = mul_acc(bufferout[7:0], b11, coeff_delay);
    Vdata <= {rout, gout, bout};
    end
    end
    else begin
    if (Vstate == 4) begin
    // double fetch
    rpart1 = mul_acc(bufferout[23:16], r11, coeff_delay);
    gpart1 = mul_acc(bufferout[15:8], g11, coeff_delay);
    bpart1 = mul_acc(bufferout[7:0], b11, coeff_delay);
    rpart2 = rpart1 - r11 + mul_acc(r11, r12, coeff_double);
    gpart2 = gpart1 - g11 + mul_acc(g11, g12, coeff_double);
    bpart2 = bpart1 - b11 + mul_acc(b11, b12, coeff_double);
    rout = clamp_value(rpart2);
    gout = clamp_value(gpart2);
    bout = clamp_value(bpart2);
    Vdata <= {rout, gout, bout};
    end
    else begin
    // non double fetch
    rpart1 = mul_acc(bufferout[23:16], r11, coeff_delay);
    gpart1 = mul_acc(bufferout[15:8], g11, coeff_delay);
    bpart1 = mul_acc(bufferout[7:0], b11, coeff_delay);
    rout = clamp_value(rpart1);
    gout = clamp_value(gpart1);
    bout = clamp_value(bpart1);
    Vdata <= {rout, gout, bout};
    end
    end
    end
    /* ----------------- function: multiply and add ----------------- */
    function [9:0]  mul_acc;
    input [7:0]    d1, d2;
    input [`COEFF_PR-1:0] c1;
    reg [`COEFF_PR+7:0]   acc;
    begin
    acc = c1 * d1 + ((1 << `COEFF_PR) - c1) * d2;
    mul_acc = acc[`COEFF_PR+7:`COEFF_PR];
    end
    endfunction
    /* function: clamp the pixel at 0 and 255 in case of over/under flow */
    function [7:0] clamp_value;
    input [9:0]  a;
    begin
    if (a[9] == 1)  clamp_value = 0;
    else if (a > 255) clamp_value = 255;
    else clamp_value = a[7:0];
    end
    endfunction
endmodule

/* -------------- multiplexor, 4 inputs, each 24 bit wide ------------- */ module
mux4by24(a,b,c,d,select,e);

    input [23:0]a,b,c,d;
    input [1:0]select;
    output [23:0]e;
    assign e = mux(a,b,c,d,select);
```

```
    function [23:0] mux;
    input [23:0] a,b,c,d;
    input [1:0]  select;
    case (select)
    2'b00: mux = a;
    2'b01: mux = b;
    2'b10: mux = c;
    2'b11: mux = d;
    endcase
    endfunction endmodule
```

B.1.3 Buffers

```
`timescale 1ns/100ps

module lbuffer(reset, clk, rd, wr, datain, dataout, full, empty);

    parameter linesize = 720;
    input reset, clk;
    input wr;  // write_enable
    input rd;  // read_signal
    input [23:0] datain;  // input data
    output [23:0] dataout; // output data
    output full;
    output empty;
    reg full, empty;
    // Define line buffer as an array
    reg [23:0] ram_data[linesize-1:0];
    // Define address bus
    reg [9:0] rd_addr;
    reg [9:0] wr_addr;
    // Define initialization for reset
    always @ (reset) begin
    if (reset) begin
    rd_addr = 0;
    wr_addr = 0;
    full = 0;
    empty = 1;
    end // if (reset)
    end
    // Read Cycle
    wire [23:0]dataout = ram_data[rd_addr[9:0]];
    always @ (posedge clk) begin
    if (!reset) begin
    if (rd == 1) begin
    rd_addr = (rd_addr + 1) % (linesize);
    empty = (rd_addr == wr_addr)? 1: 0;
    if (full == 1) full = 0;
    end // if (rd == 1)
    end
    end
    // Write Cycle
    always @ (posedge clk ) begin: do_write
    if (!reset) begin
    if (wr == 1) begin
    ram_data[wr_addr[9:0]] = datain;
    wr_addr = (wr_addr + 1) % (linesize);
    full = (wr_addr == rd_addr) ? 1:0;
```

```
    if (empty == 1) empty = 0;
    end // if (wr == 1)
    end // if (!reset)
    end
endmodule // lbuffer

module buffer(reset, clk, rd, wr, datain, dataout, full, empty);

    parameter linesize = 8;
    input reset, clk;
    input wr;  // write_enable
    input rd;  // read_signal
    input [23:0] datain;  // input data
    output [23:0] dataout; // output data
    output full;
    output empty;
    reg full, empty;
    // Define line buffer as an array
    reg [23:0] ram_data[linesize-1:0];
    // Define address bus
    reg [9:0] rd_addr, rd_addr1, rd_addr2;
    reg [9:0] wr_addr, wr_addr1, wr_addr2, wr_addr3;
    // Define initialization for reset
    always @ (reset) begin
    if (reset) begin
    rd_addr = 0;
    wr_addr = 0;
    full = 0;
    empty = 1;
    end
    end
    // Read Cycle
    wire [23:0]dataout = ram_data[rd_addr[9:0]];
    // empty signal goes up one cycle before it's actually empty
    always @ (posedge clk) begin
    if (!reset) begin
    if (rd == 1) begin
    rd_addr  = (rd_addr + 1) % (linesize);
    end
    rd_addr1 = (rd_addr) % (linesize);
    rd_addr2 = (rd_addr + 1) % (linesize);
    empty = (rd_addr1 == wr_addr || rd_addr2 == wr_addr) ? 1: 0;
    end
    end
    // Write Cycle, full signal goes up two cycles before it's actually full
    always @ (posedge clk ) begin: do_write
    if (!reset) begin
    if (wr == 1) begin
    ram_data[wr_addr[9:0]] = datain;
    wr_addr  = (wr_addr + 1) % (linesize);
    end // if (wr == 1)
    wr_addr1 = (wr_addr + 1) % (linesize);
    wr_addr2 = (wr_addr + 2) % (linesize);
    wr_addr3 = (wr_addr + 3) % (linesize);
    full = (wr_addr1 == rd_addr || wr_addr2 == rd_addr ||
    wr_addr3 == rd_addr) ? 1:0;
    end
    end
endmodule
```

B.1.4 Scaler

```
`timescale 1ns/100ps `define COEFF_PR 6  // number of bit of precision for the
filter coefficients `define SPACE_PR 13 // number of bit of precision for the
increment factor

module scaler(clk, reset, winAct, dval1, dval2, rd_data1, rd_data2, datain1,
datain2, rH, rV, OLDXSIZE, NEWXSIZE, OLDYSIZE, NEWYSIZE, Vstate, Vreset, scan_end,
Hdataout, Hdata, last_line);

    inputclk;
    inputreset;
    inputdval1, dval2;  // pipeline data valid //   input winAct;
    input [23:0]datain1;       // data from pipeline 1
    input [23:0]datain2;       // data from pipeline 2
    input [`SPACE_PR:0]rH;
    input [`SPACE_PR:0]rV;
    input [10:0]OLDXSIZE;
    input [10:0]OLDYSIZE;
    input [10:0]NEWXSIZE;
    input [10:0]NEWYSIZE;
    /* --------------------- module instantiation ----------------------- */
    wireVmode = (NEWYSIZE >= OLDYSIZE) ? 1 : 0;
    wireHmode = (NEWXSIZE >= OLDXSIZE) ? 1 : 0;
    wirebuffer_empty;
    wirebuffer_full;
    wirestall = 0;
    wireVdata_ready;
    wire [23:0]Vdata;
    outputHdataout;
    wireHdataout;
    outputHdata;
    wire [23:0]Hdata;
    wirerd_buffer, wr_buffer;
    wire [23:0]Vdata_o;
    regVdata_i;
    outputrd_data1;
    outputrd_data2;
    wirerd_data1;
    wirerd_data2;
    output [2:0]Vstate;
    wire [2:0]Vstate;
    outputscan_end;
    wirescan_end;
    outputVreset;
    wireVreset;
    outputlast_line;
    wirelast_line;
    Vmodule V(clk, reset, rd_data1, rd_data2, Vmode, buffer_full,
    dval1, dval2, datain1, datain2,
    Vdata_ready, Vdata, rV, NEWYSIZE, OLDYSIZE, OLDXSIZE, Vstate,
    Vreset, scan_end);
    Hmodule H(clk, reset, stall, Hstall, Hmode, Vdata_i, Vdata_o,
    Hdataout, Hdata, rH, NEWXSIZE, OLDXSIZE, last_line, Vreset);
    buffer b(reset, clk, rd_buffer, wr_buffer, Vdata, Vdata_o,
    buffer_full, buffer_empty);
    assign wr_buffer = Vdata_ready & ~reset;
    assign rd_buffer = ~Hstall & ~reset;
    always @ (posedge clk) Vdata_i = ~buffer_empty;
endmodule
```

B.2 Test Drivers

B.2.1 Driver for Horizontal Unit

```
`timescale 1ns/100ps `define COEFF_PR 6  // number of bit of precision for the
filter coefficients `define SPACE_PR 13 // number of bit of precision for the
increment factor

module test_horiz;

    parameter       CYCLE = 10;
    parameter       HALFCYCLE = CYCLE/2;
    parameter       NORMD = `SPACE_PR - `COEFF_PR;
    parameter       NORM = 1 << `SPACE_PR;
    reg       clk, stall, hmode, reset, Vdataout;
    reg [`SPACE_PR:0] rh;// increment factor
    reg [10:0]          NEWXSIZE, OLDXSIZE;
    reg [23:0]       Vdata;
    wire [23:0]       Hdata;
    wire                Hdataout, Hstall;
    Hmodule H(clk,
    reset,
    stall,
    Hstall,
    hmode,
    Vdataout,
    Vdata,
    Hdataout,
    Hdata,
    rh,
    NEWXSIZE,
    OLDXSIZE,
    last_line,
    Vreset);
    // Define clock
    initial clk = 0;
    always #HALFCYCLE clk = ~clk;
    initial begin
    NEWXSIZE = 8;
    OLDXSIZE = 8;
    // calculating increment factor
    rh = (NEWXSIZE >= OLDXSIZE) ? (NORM * (OLDXSIZE - 1) / (NEWXSIZE - 1)) :
    (NORM * NEWXSIZE / OLDXSIZE);
    // error correction
    if (rh == NORM) rh = rh;
    else if (rh >= (1 << (`SPACE_PR - 1))) rh = rh + 1;
    stall = 0;
    hmode = (NEWXSIZE >= OLDXSIZE) ? 1 : 0;
    reset = 1;
    Vdataout = 1;
    Vdata = 127;
    #10 reset = 0;
    //    #40 stall = 1;
    //    #20 stall = 0;
    #1500 $finish;
    end
    // Define data input for testing
```

```
    always @(posedge clk) begin
    Vdata = Vdata + 16;
    if (Vdata > 255) Vdata = Vdata - 256;
    end
endmodule
```

B.2.2 Driver for Vertical Unit

```
module test_vertical;

    parameter       CYCLE = 10;
    parameter       HALFCYCLE = CYCLE/2;
    parameter       NORMD = `SPACE_PR - `COEFF_PR;
    parameter       NORM = 1 << `SPACE_PR;
    reg       clk, stall, vmode, reset, data_val1, data_val2;
    reg [`SPACE_PR:0] rv;
    reg [10:0]       NEWYSIZE, OLDYSIZE, OLDXSIZE;
    reg [23:0]       data1, data2;
    wire [23:0]       Vdata;
    reg       Hstall;
    wire       Vdataout;
    wire       rd_data1, rd_data2;
    wire [2:0]       Vstate;
    wire       Vreset, scan_end;

    Vmodule V(clk, reset, rd_data1, rd_data2, vmode, Hstall, data_val1, data_val2,
data1, data2, Vdataout, Vdata, rv, NEWYSIZE, OLDYSIZE, OLDXSIZE, Vstate, Vreset,
scan_end);

    // Define clock
    initial clk = 0;
    always #HALFCYCLE clk = ~clk;
    initial begin
    NEWYSIZE = 9;
    OLDYSIZE = 9;
    OLDXSIZE = 5;
    rv = (NEWYSIZE >= OLDYSIZE) ? (NORM * (OLDYSIZE - 1) / (NEWYSIZE - 1)) :
    (NORM * NEWYSIZE / OLDYSIZE);
    if (rv == NORM) rv = rv;
    else if (rv >= (1 << (`SPACE_PR - 1))) rv = rv + 1;
    Hstall = 0;
    data1 = 0;
    data2 = 127;
    stall = 0;
    data_val1 = 1;
    data_val2 = 1;
    vmode = (NEWYSIZE >= OLDYSIZE) ? 1 : 0;
    reset = 1;
    #15 reset = 0;
    #1500 $finish;
    end
    // Define input data for testing
    always @(posedge clk) begin
    data1 <=#1 data1 + 1;
    data2 <=#1 data2 + 1;
    if (data1 > 255) data1 = data1 - 256;
    if (data2 > 255) data2 = data2 - 256;
```

```
end endmodule
```

B.2.3 Driver for buffer

```
module test_lbuffer;

   reg clk;
   reg reset;
   reg   wr, rd;
   reg [23:0] datain;
   wire [23:0] dataout;
   wire  full, empty;
   // Instantiate Line Buffer
   lbuffer b1(reset, clk, rd, wr, datain, dataout, full, empty);
   // Define clk
   initial clk = 0;
   always #5 clk = ~clk;
   // Reset Line Buffer
   initial
   begin
      reset = 1;
      #5 reset = 0;
      #600 $finish;
   end
   // Define Data Input
   initial datain = 0;
   always @ (posedge clk) begin
      datain = datain + 1;
   end
   // Write Operation
   initial
   begin
      #10 wr = 1;
      #80 wr = 0;
      #30 wr = 1;
      #190 wr = 0;
   end
   // Read Operation
   initial
   begin
      rd = 0;
      #50 rd = 1;
      #70 rd = 0;
      #210 rd = 1;
   end
   // monitoring the output
   always @ (posedge clk)
      $strobe(rd, wr, dataout, datain, empty, full);
endmodule
```

B.2.4 Driver for scaler

```
module test_scaler;
```

```
    parameter       NORM = 1 << `SPACE_PR;
    integer       i, fd;
    reg       clk;
    reg       reset;
    reg       winAct;
    reg       dval1, dval2;
    reg [23:0]       datain1, datain2;
    reg [`SPACE_PR:0] rV, rH;
    reg [10:0]       OLDXSIZE;
    reg [10:0]       OLDYSIZE;
    reg [10:0]       NEWXSIZE;
    reg [10:0]       NEWYSIZE;
    wire [23:0]       dataout;
    wire       rd_data1, rd_data2;
    reg [23:0]       image[200000:0];
    wire [2:0]       Vstate;
    wire       Vreset;
    wire       scan_end;
    wire       Hdataout;
    wire [23:0]       Hdata;
    wire       last_line;

    scaler s(clk, reset, winActive, dval1, dval2, rd_data1, rd_data2, datain1,
datain2, rH, rV, OLDXSIZE, NEWXSIZE, OLDYSIZE, NEWYSIZE, Vstate, Vreset, scan_end,
Hdataout, Hdata, last_line);

    initial clk = 0;
    always #5 clk = ~clk;
    initial begin
    $readmemh("flower.txt", image);
    OLDXSIZE = image[0];
    OLDYSIZE = image[1];
    NEWXSIZE = image[2];
    NEWYSIZE = image[3];
    i = 4;
    datain1 = 0;
    datain2 = 0;
    // prepare output
    fd = $fopen("fout.txt");
    $fwrite(fd, NEWXSIZE, " ", NEWYSIZE, "\n");
    rV = (NEWYSIZE >= OLDYSIZE) ?
    (NORM * (OLDYSIZE - 1) / (NEWYSIZE - 1)) :
    (NORM * NEWYSIZE / OLDYSIZE);
    if (rV == NORM) rV = rV; // orginal size
    else if (rV >= (1 << (`SPACE_PR - 1))) rV = rV + 1;
    rH = (NEWXSIZE >= OLDXSIZE) ?
    (NORM * (OLDXSIZE - 1) / (NEWXSIZE - 1)) :
    (NORM * NEWXSIZE / OLDXSIZE);
    if (rH == NORM) rH = rH; // original size
    else if (rH >= (1 << (`SPACE_PR - 1))) rH = rH + 1;
    dval1 = 1; // assuming data always available
    dval2 = 1;
    reset = 1;
    #15 reset = 0;
    end
    // define end of simulation
    always @ (last_line) begin
    if (!last_line && !reset) #100 begin
    $fclose(fd);
```

```
    $finish;
    end
    end
    // Define input data for testing
    always @(negedge clk) begin
    if (rd_data1) begin
    datain1 <= image[i];
    if (rd_data2)
    datain2 <= image[i+OLDXSIZE];
    else datain2 <= datain2;
    i = i+1;
    end
    else begin
    datain1 = datain1;
    datain2 = datain2;
    end
    end
    // address control
    always @ (scan_end) begin
    if (scan_end) begin
    if (Vstate == 1) i = i - OLDXSIZE;      // refetch the same line
    else if (Vstate == 4) i = i + OLDXSIZE; // double fetch
    end
    end
    // output data to file
    always @(negedge clk) begin
    if (Hdataout)
    $fwrite(fd, Hdata, " ");
    end
endmodule

// simple testing for small numbers

module testbench;

    parameter       NORM = 1 << `SPACE_PR;
    integer       i, fd;
    reg       clk;
    reg       reset;
    reg       winAct;
    reg       dval1, dval2;
    reg [23:0]       datain1, datain2;
    reg [`SPACE_PR:0] rV, rH;
    reg [10:0]       OLDXSIZE;
    reg [10:0]       OLDYSIZE;
    reg [10:0]       NEWXSIZE;
    reg [10:0]       NEWYSIZE;
    reg [23:0]       image[1000:0];
    wire [23:0]       dataout;
    wire       rd_data1, rd_data2;
    wire [2:0]       Vstate;
    wire       Vreset;
    wire       scan_end;
    wire       Hdataout;
    wire [23:0]       Hdata;
    wire       last_line;

    scaler s(clk, reset,winActive, dval1, dval2, rd_data1, rd_data2, datain1,
datain2, rH, rV, OLDXSIZE, NEWXSIZE, OLDYSIZE, NEWYSIZE, Vstate, Vreset, scan_end,
```

```
Hdataout, Hdata, last_line);

    initial clk = 0;
    always #5 clk = ~clk;
    initial begin
    OLDXSIZE = 9;
    OLDYSIZE = 9;
    NEWXSIZE = 9;
    NEWYSIZE = 9;
    datain1 = 0;
    datain2 = 127;
    // calculation r factor
    rV = (NEWYSIZE > OLDYSIZE) ?
    (NORM * (OLDYSIZE - 1) / (NEWYSIZE - 1)) :
    (NORM * NEWYSIZE / OLDYSIZE);
    // error correction
    if (rV == NORM) rV = rV;
    else if (rV >= (1 << (`SPACE_PR - 1))) rV = rV + 1;
    rH = (NEWXSIZE > OLDXSIZE) ?
    (NORM * (OLDXSIZE - 1) / (NEWXSIZE - 1)) :
    (NORM * NEWXSIZE / OLDXSIZE);
    // error correction
    if (rH == NORM) rH = rH;
    else if (rH >= (1 << (`SPACE_PR - 1))) rH = rH + 1;
    // assuming data always available for now
    dval1 = 1;
    dval2 = 1;
    reset = 1;
    #15 reset = 0;
    #1500 $finish;
    end
    // test data
    always @ (posedge clk) begin
    datain1 = datain1 + 2;
    datain2 = datain2 + 2;
    end
endmodule
```

Appendix C

Simulation Results

This section includes some example of the wave files resulted from simulating the Verilog code using small line sizes. The order of the test cases shown here are:

C.1 Hmodule (Horizontal Scaler)

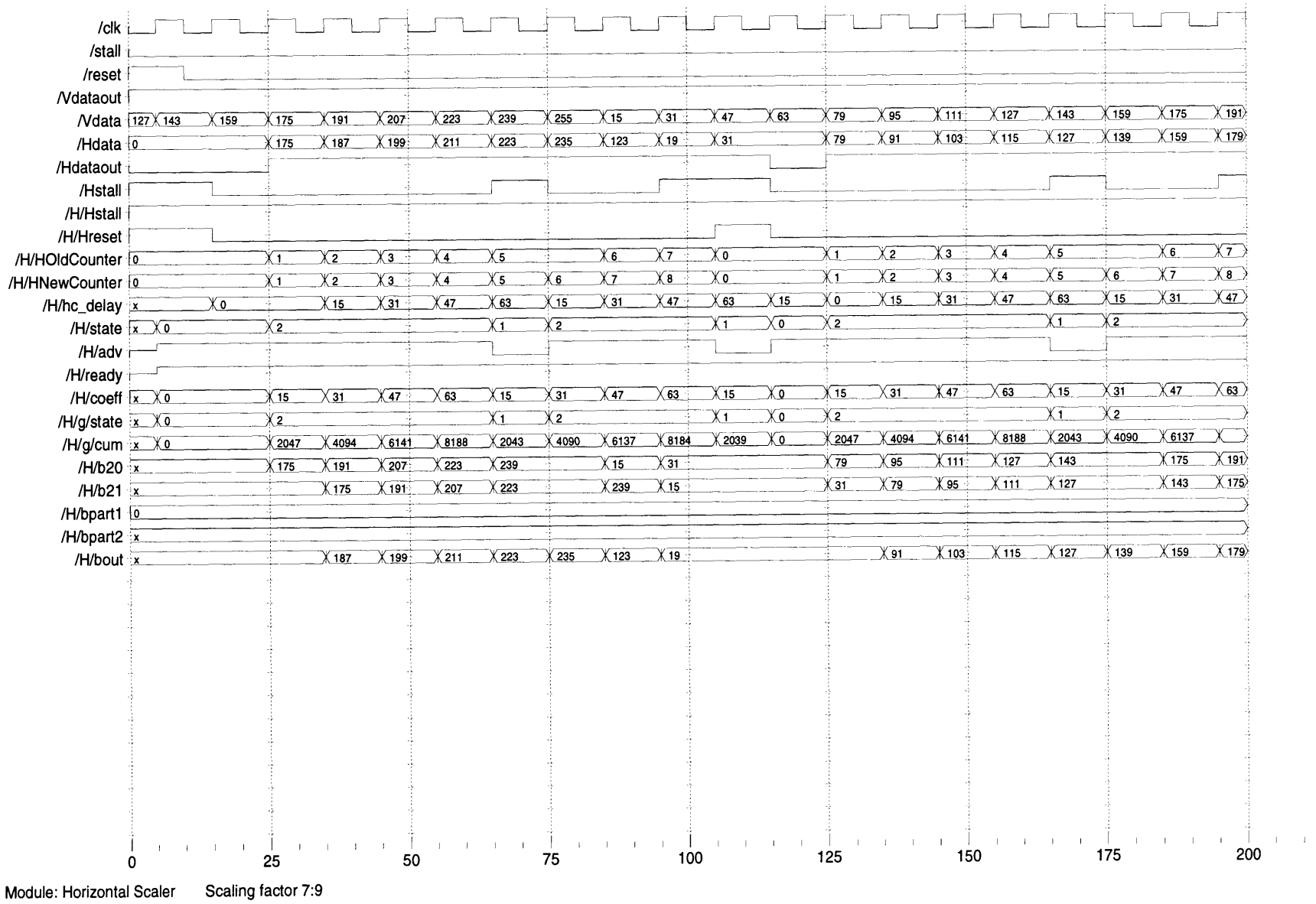
1. Scaling factor 7:9
2. Scaling factor 9:4

C.2 Vmodule (Vertical Scaler)

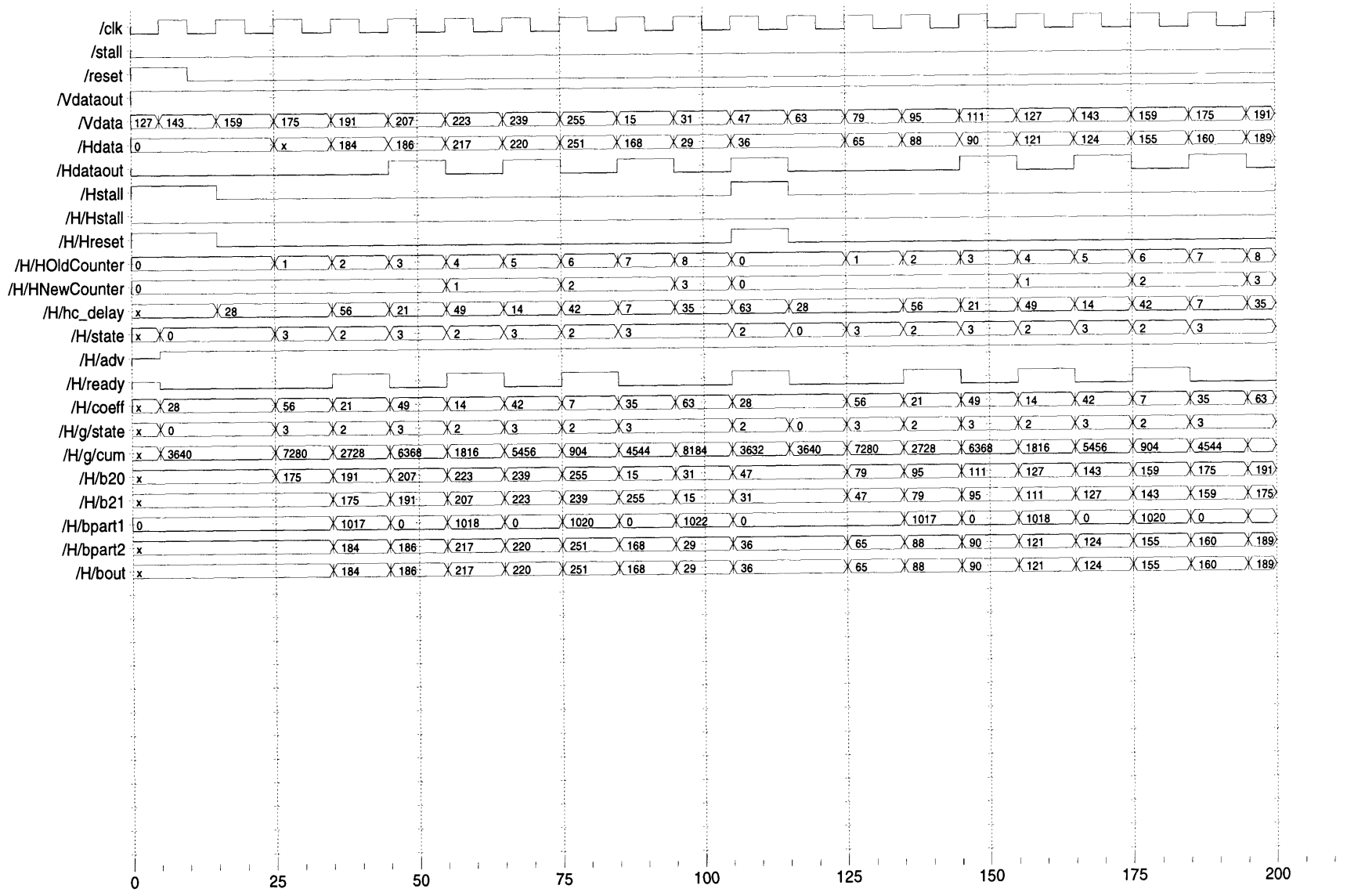
1. Scaling factor 4:9
2. Scaling factor 9:7

C.3 Scaler (Full Scaler)

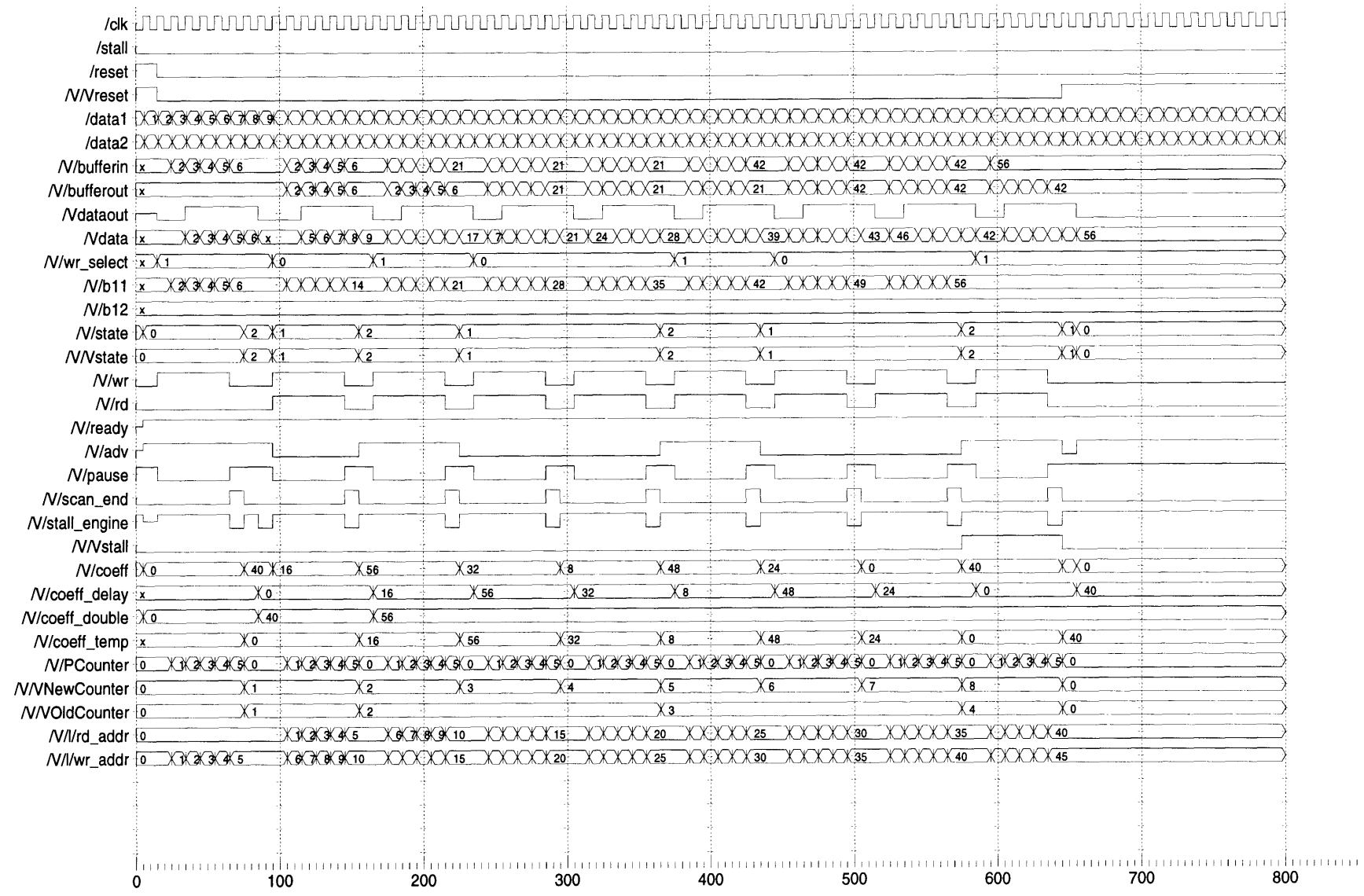
1. Horizontal scaling factor 4:9, vertical scaling factor 7:5



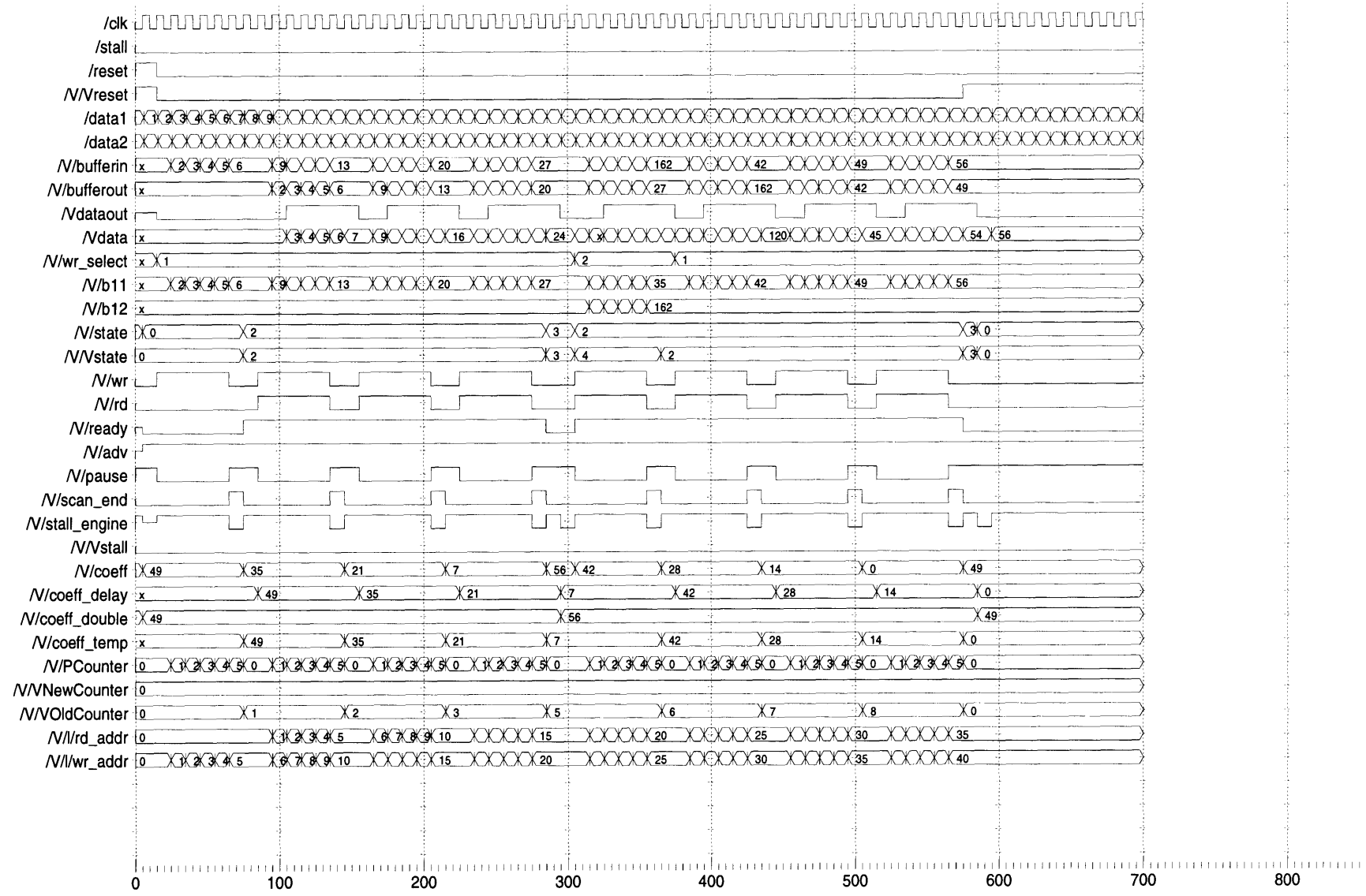
Module: Horizontal Scaler Scaling factor 7:9



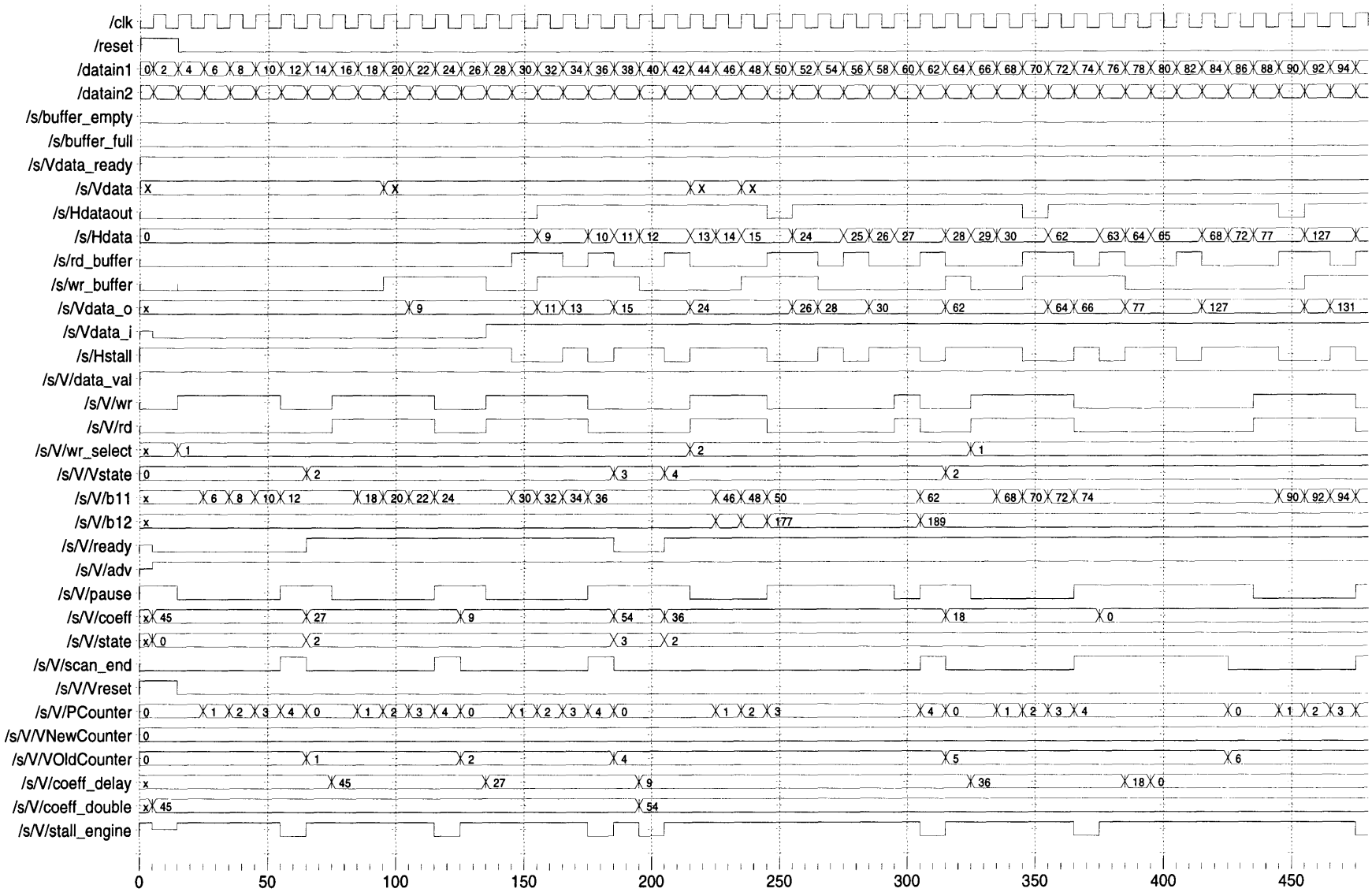
Module: Horizontal Scaler Scaling factor 9:4



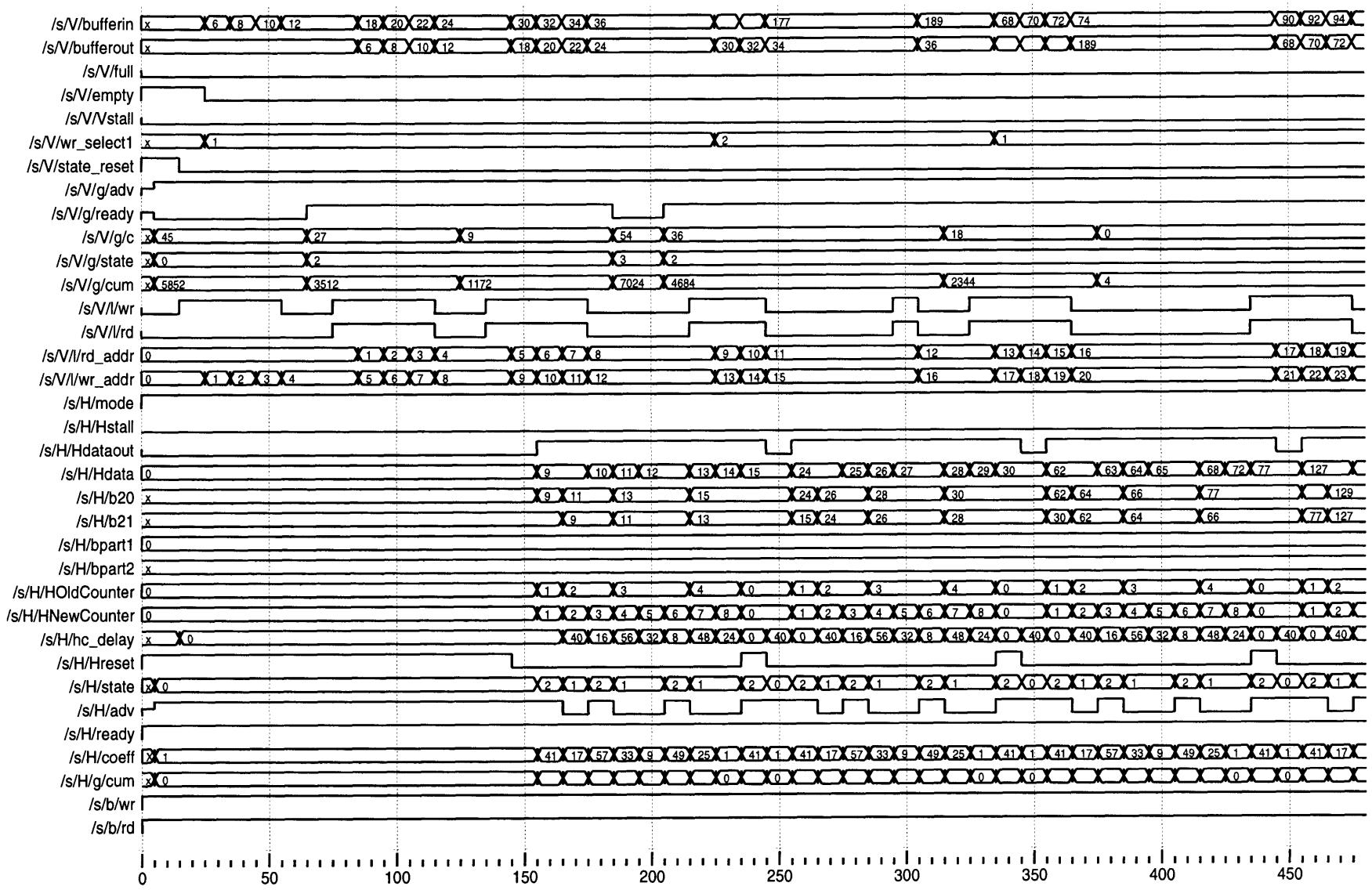
Module: Vertical Scaler Scaling factor 4:9



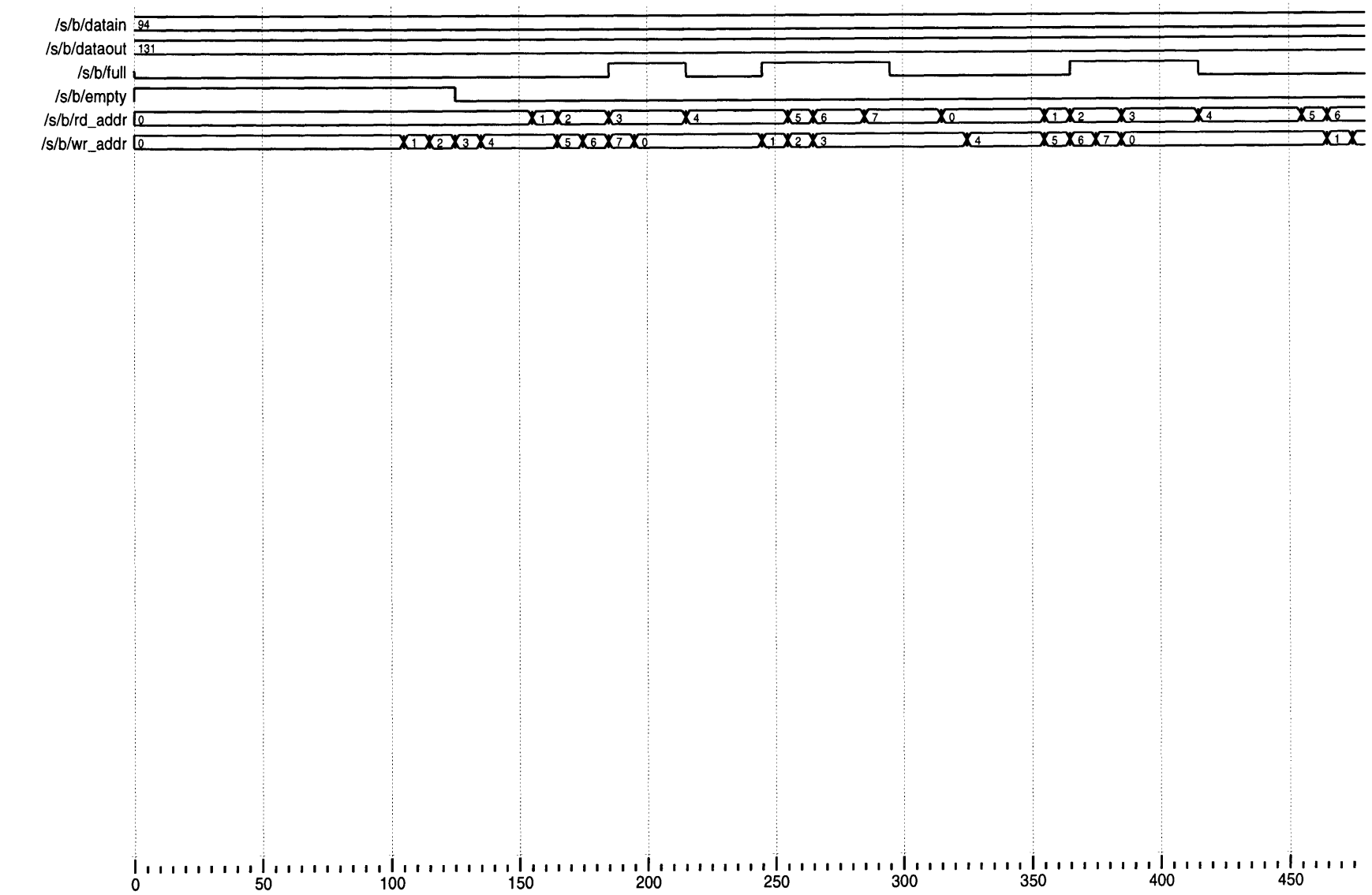
Module: Vertical Scaler Scaling factor 9:7



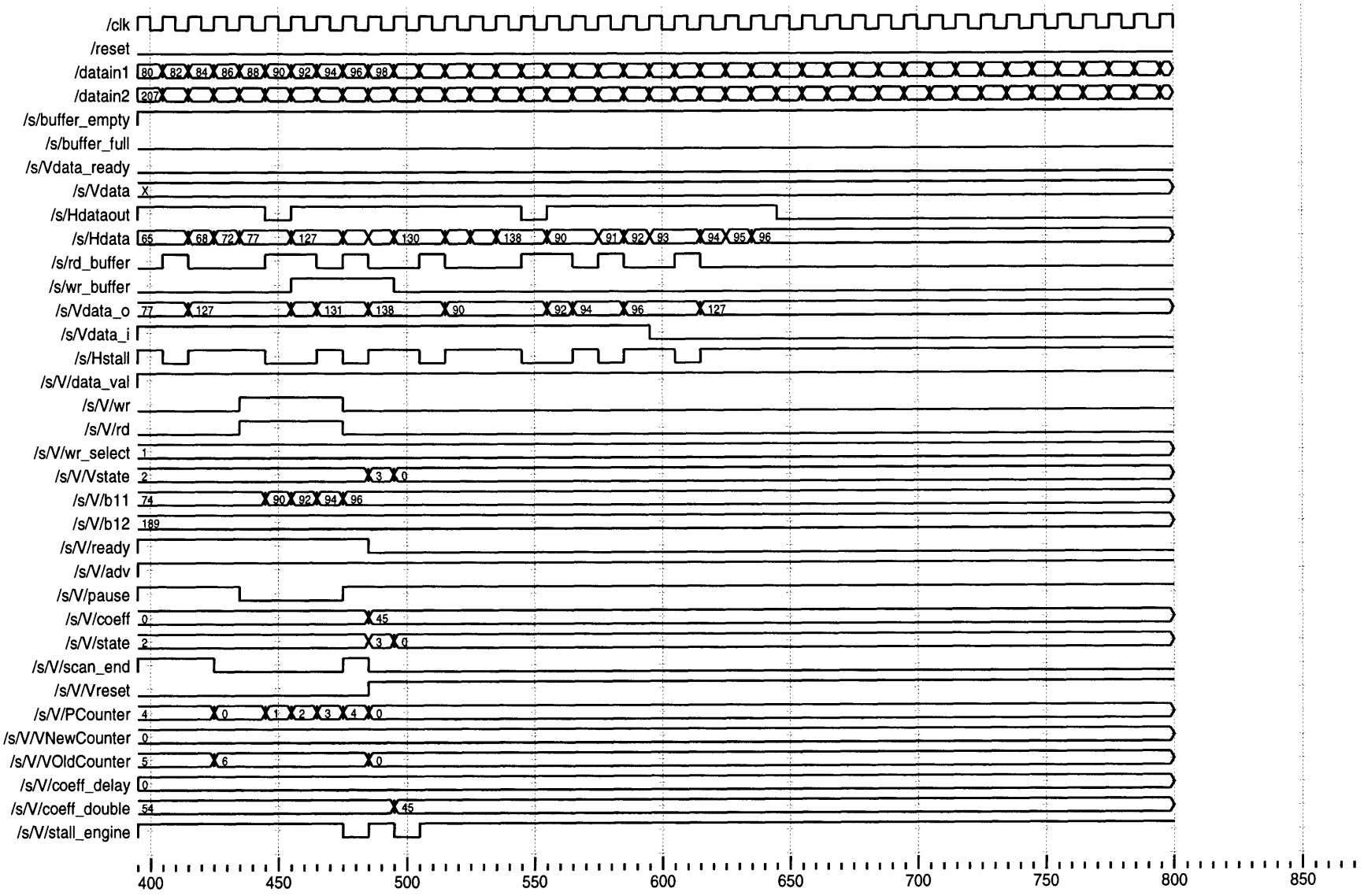
Module: Full Scaler Horizontal scaling factor 4:9 Vertical scaling factor 7:5 Time frame: 0..400 Page 1.1



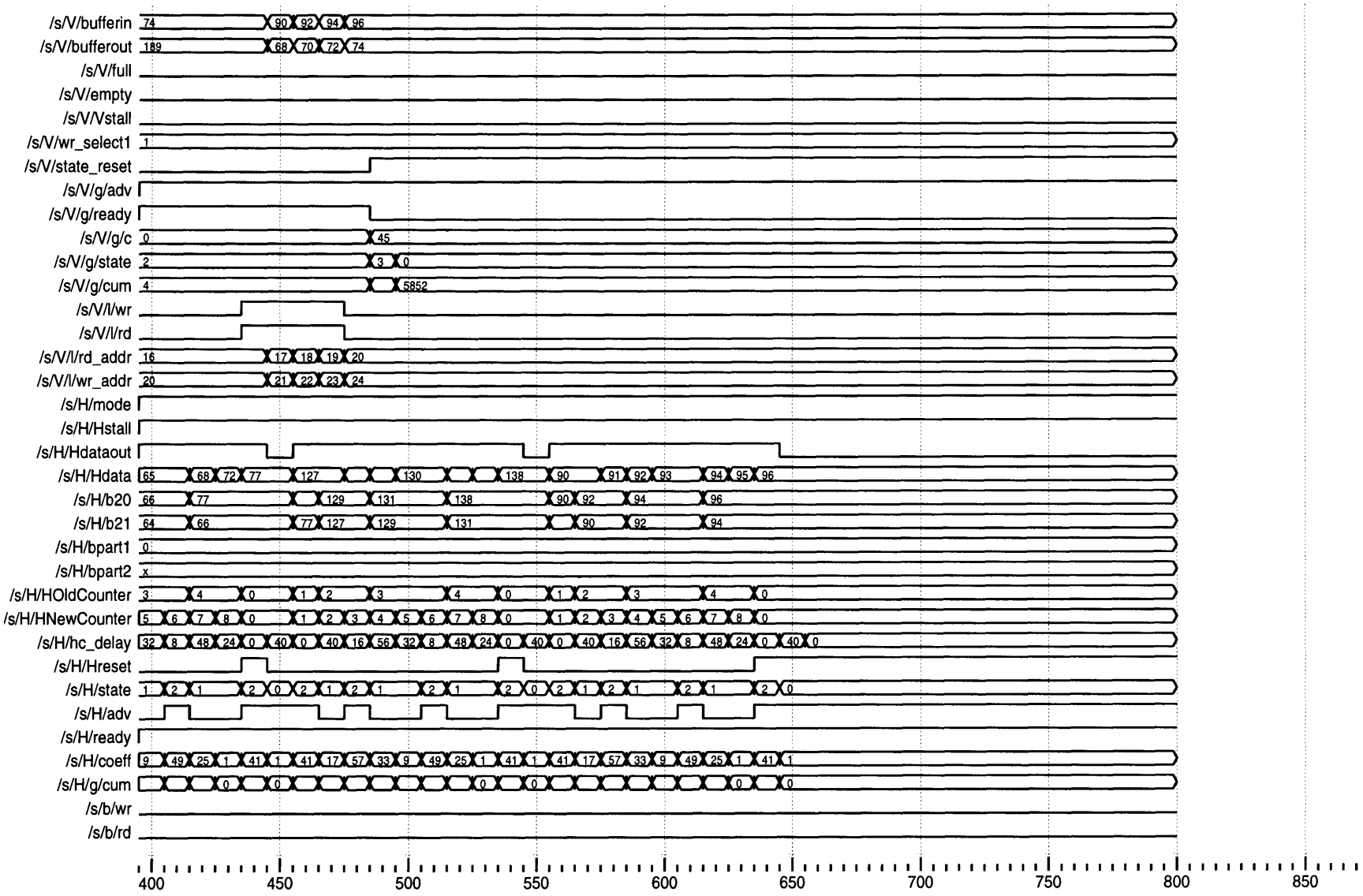
Module: Full Scaler Horizontal scaling factor 4:9 Vertical scaling factor 7:5 Time frame: 0..400 Page 1.2



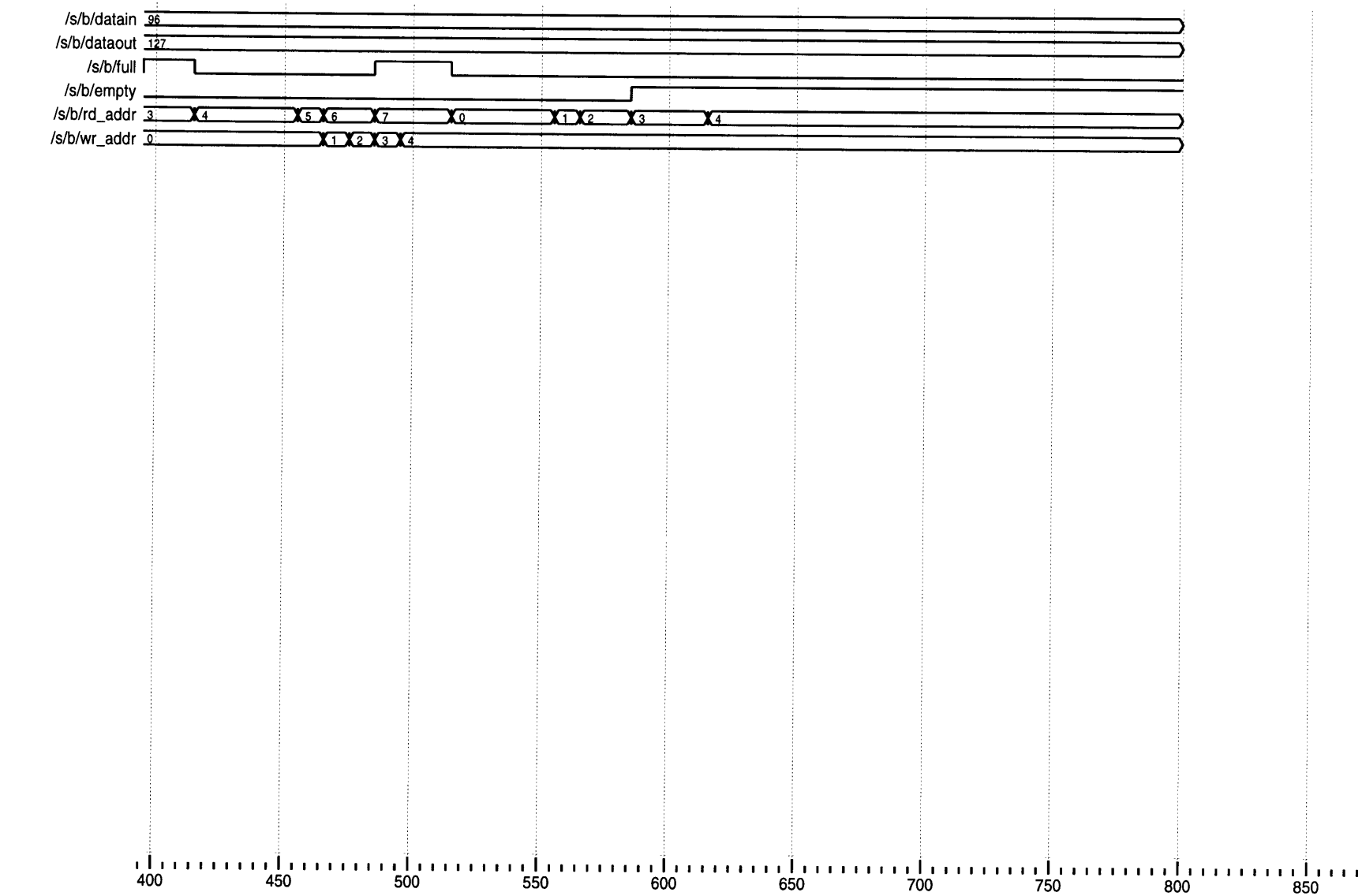
Module: Full Scaler Horizontal scaling factor 4:9 Vertical scaling factor 7:5 Time frame: 0..400 Page 1.3



Module: Full Scaler Horizontal scaling factor 4:9 Vertical scaling factor 7:5 Time frame: 400..800 Page 2.1



Module: Full Scaler Horizontal scaling factor 4:9 Vertical scaling factor 7:5 Time frame: 400..800 Page 2.2



Module: Full Scaler Horizontal scaling factor 4:9 Vertical scaling factor 7:5 Time frame: 400..800 Page 2.3

References

[1] R.E. Crochiere and L.R. Rabiner, *Multirate Digital signal Processing*, Englewood Cliffs, NJ: Prentice-Hall 1983.

[2] A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, NJ: Prentice-Hall 1983.

[3] D. Kirk, *Graphics gems III*, Harcourt Brace Jovanovich, 1992.

[4] J. Foley, A. van Dam, S. Feiner, J. Hughes, *Computer Graphics: Principles and Practice,* Addison-Wesley Publishing Company, Inc. 1990.

[5] L.D. Seiler and R.A. Ulichney, "Integrating Video Rendering into Graphics Accelerator Chips," *Digital Technical Journal*, Vol. 7 No. 4 1995.

[6] J.S. Lim, *Two-dimensional Signal and Image Processing, Englewood Cliffs*, NJ: Prentice-Hall 1990.

[7] W.K. Pratt, *Digital Image Processing*, Wiley, New York 1991.

[8] R.M.P. West, "Computer Graphics - Getting the Data Out of the Machine and Onto the Screen," SID'96 Seminar Notes, Seminar M7.